View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ELTE Digital Institutional Repository (EDIT)

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatikai kar

Algoritmusok és Alkalmazásaik Tanszék

Online puzzle játék

Kovácsné Pusztai Kinga
egyetemi tanársegéd

Gyetvai Gábor
programtervező informatikus BSc

Budapest, 2018

Tartalom

1 Bevezetés

Szakdolgozatom témájaként egy online kirakós játékot választottam, mivel ez a típusú játék már gyerekkorom óta érdekel. Mindig szerettem volna online játék fejlesztését kipróbálni, ezért úgy gondoltam, hogy összekötöm az informatika tanulmányaim lezárását szolgáló szakdolgozatommal. Témaválasztásomat befolyásolta az is, hogy a munkám során ugyan a felhasznált technológiák egy részét már használtam, azonban az online puzzle játék elkészítéséhez számos új területtel is meg kellett ismerkednem. Ezek közül kiemelném a Typescript-et, Asp.Net Core-t, az Angulart, illetve a Postgres adatbázist.

Dolgozatom az elvártak szerint két fő területre osztom. Az első felében a felhasználói dokumentációt vezetem végig, az alkalmazás telepítésétől, a különböző fő funkciók használatán át, egészen a képkategóriákig. A funkciók leírása során természetesen kitérek a felmerülő hibaüzenetekre is és lehetséges megoldásaikra. A munkám másik jelentős eleme a fejlesztői dokumentáció. Ebben a részben elemzem az alkalmazás tervezett felépítését, és rétegeit, valamint a játék alkalmazás részleteit, tesztek leírását, és a forrásmegjelölést. A játék alkalmazás részletei kitérnek az elemek alapformájára, a maszkolására, és logikájára, valamint ezek implementációjára. Ebben a részben szintén részletezem a játék logikáját, illetve a statisztikák működését is.

Köszönetet mondok Kovácsné Pusztai Kinga tanárnőnek, hogy elvállalta szakdolgozatom témavezetését. Jó tanácsaival és észrevételeivel hozzájárult a szakdolgozatom megvalósulásához.

2 Felhasználói dokumentáció

2.1 Az alkalmazás telepítése és teszt futtatása

2.1.1 Az alkalmazás telepítése

A telepítés leírt lépései egy teszt telepítést írnak Ubuntu 18.04 operációs rendszerre. Ha az alkalmazást más rendszerre vagy nem tesztelés céljából telepítjük, akkor további lépések is szükségesek lehetnek.
A telepítéshez szükség van a docker, illetve docker-compose alkalmazásokra, amelyeket a következő módon telepíthetünk:

```
sudo apt install docker.io
sudo apt install docker-compose
```

Ezután hozzunk létre egy mappát a nekünk tetsző névvel, majd másoljuk bele a mellékelt `justapuzzle.tar` és `docker-compose.test.yml` fájlokat. Töltsük be a docker image-et a következő utasítással:

```
docker load -i justapuzzle.tar
```

A betöltött image id-ját le kell kérdeznünk, amit a következő utasítással lehet megtenni:

```
docker image ls
```

A kapott adatokból az első sor az érdekes, annak is az `IMAGE ID` oszlopa, és ezzel az id-val taggeljük meg az alkalmazás image-t.

```
docker image tag <a kiolvasott image id> justapuzzle:final
```

2.1.2 Az alkalmazás elindítása

Navigáljunk vissza az előzőleg létrehozott mappába, és adjuk ki a

```
docker-compose -f docker-compose.yml up
```

parancsot. Ez elindítja az alkalmazást, ami rövid idő eltelte után elérhető lesz a `localhost:5000` url-en.

Ezzel egy időben létre jönnek a következő mappák:

- `data` – ide kerülnek az adatbázis adat fájlai
- `logs` – ide fognak kerülni a keletkező log-ok
- `mails` – ide fognak kerülni az alkalmazás által küldött email-ek

Navigáljunk el a `localhost:5000` url-re, ha ezt még nem tettük meg és regisztráljunk egy új felhasználót. A megerősítő email a `mails` mappába fog kerülni a fájl neve pedig a `<email cím @ nélkül><random generált karakterek>.txt` mintát fogja követni. Ebben a fájlban található az email cím megerősítéséhez szükséges link, amit másoljunk be a böngésző címsorába.
Miután megtörtént a megerősítés, az új felhasználó be tud majd lépni. Ekkor még üres az adatbázis, emiatt nem találhatóak még meg a kategóriák, illetve a játék során használható képek sem. Ahhoz, hogy ezek elérhetőek legyenek, a friss felhasználónak adjunk alkalmazás adminisztrátor jogokat.
Ehhez a következő lépések szükségesek:

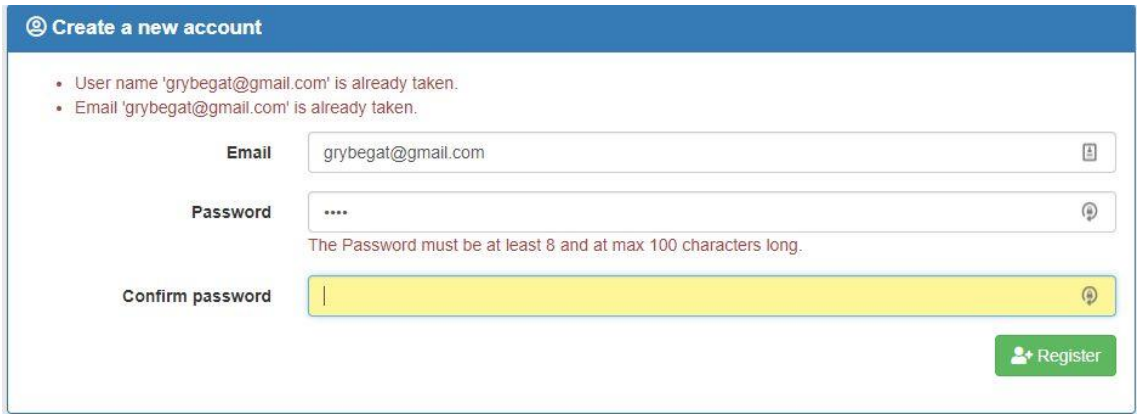
- Kapcsolódjunk a futó adatbázis docker konténerhez:
  `docker exec -tiu postgres justapuzzlepostgres psql`
- Kapcsolódjunk a PuzzleDb adatbázishoz:
  `\connect PuzzleDb;`
- Futtassuk le a következő script-et:
  ```
  insert into "AspNetUserRoles" (select (select "Id" from
  "AspNetUsers" where "Email" = '<regisztrált felhasználó
  email címe') as "UserId", (select "Id" from "AspNetRoles"
  where "Name" = 'Admin') as "RoleId");
  ```
- Lépjünk ki és be a felhasználóval és ekkor az adminisztrációs felületen fogjuk találni magunkat.

A kategóriák és képek felvételét a felhasználói dokumentáció további fejezetei írják majd le. Az alkalmazás leállítható a `docker-compose -f docker-compose.test.yml down` parancs kiadásával.

2.2 Regisztráció

Az alkalmazás url-jére navigálva a belepési képernyőn fogjuk találni magunk, ahol a `Create new account` linkre kattintva lehet új felhasználót regisztrálni.
Az Email mezőben meg kell adni egy érvényes email címet, amit el is érünk. A Password mezőbe meg kell adni a jelszót, ami tartalmaz legalább egy nagy betűt, egy kisbetűt és egy számot. A jelszó minimális hossza 8 karakter és a maximális hossza pedig 100 karakter. A Confirm password mezőben újra meg kell adnunk jelszót, aminek egyeznie kell az előző mezőben megadotttal.
Ezután kattintsunk a `Register` gombra. Ha regisztráció sikeres, akkor az alkalmazás átirányít a megerősítő email elküldve oldalra. Ekkor ellenőrizni kell a regisztrációnál megadott email címet és kapott emailben lévő linkre kell kattintani. Ezután az email cím megerősítés sikeres oldalra jutunk, ahol a `Click here to login` linkre kattintva a belépési oldalra jutunk újra és itt a megadott email cím jelszó kombinációval be tudunk lépni.

2.2.1 Hibaüzenetek a regisztrációnál



1. ábra Hibaüzenetek megjelenése

Hibaüzenet	Hibaüzenet oka
The Email field is required	Nem adott meg email címet
The Password field is required	Nem adott meg jelszót
User name '<email cím>' is already taken.	Egy már regisztrált felhasználó email címét adta meg.
Email '<email cím>' is already taken.	Egy már regisztrált felhasználó email címét adta meg.

Hibaüzenet	Hibaüzenet oka
The Email field is not a valid email address.	Nem email címet írt az Email mezőbe.
The password must contain at least 1 Uppercase and 1 lowercase character, plus 1 digit!	A megadott jelszó nem tartalmaz legalább 1 nagybetűt, 1 kisbetűt és 1 számot.
The password must be at least 8 and at max 100 characters long.	A megadott jelszó kevesebb 8 karakternél vagy több mint 100 karakter.
The password and confirmation password do not match.	A Confirm password mezőbe írt érték nem egyezik meg a Password mezőbe írt értékkel.

2.3 Belépés

A belépési oldalon az email cím és a jelszó megadása után a 'Login' gombra kattintva tudunk belépni az alkalmazásba. Amennyiben a belépő felhasználó Játékos (`Player`) szerepkörű, akkor az alkalmazás a játékosok kategória böngésző oldalára irányítja. Alkalmazás Adminisztrátor (`Admin`) szerepkör esetén az adminisztrátorok kategória-böngésző oldalára jut a felhasználó.

2.3.1 Hibaüzenetek a belépésnél

Hibaüzenet	Hibaüzenet oka
The Email field is required.	Nem adott meg email címet.
The Password field is required.	Nem adott meg jelszót.
The Email field is not a valid e-mail address.	Nem email címet adott meg az Email mezőben.
Invalid login attempt.	Ennek több oka is lehetséges: nem regisztrált felhasználó email címét adta meg, vagy a megadott email és jelszó páros nem egyezik regisztrált felhasználó adataival.

2.4 Elfelejtett jelszó

A belépési oldalon található `Forgot your password?` linkre kattintva jutunk az elfelejtett jelszó oldalra, ahol egy email cím megadása után az alkalmazás küld egy emailt, amiben a jelszó megváltoztatásához szükséges link található. Erre a linkre kattintva jutunk el az oldalra, ahol meg tudjuk változtatni a jelszót. Itt ki kell tölteni az Email, Password és Confirm password mezőket, majd kattintani a `Reset` gombra. Ha jelszó változtatás sikeres, akkor egy megerősítő oldalra jutunk, ahol a `Click here to login` link a belépési oldalra navigál.

2.4.1 Hibaüzenetek az elfelejtett jelszónál

Hibaüzenet	Hibaüzenet oka
The Email field is required.	Nem adott meg email címet.
The Email field is not a valid email address.	Nem email címet írt az Email mezőbe.
The password must be at least 8 and at max 100 characters long.	A megadott jelszó kevesebb 8 karakternél vagy több mint 100 karakter.
The password and confirmation password do not match.	A Confirm password mezőbe írt érték nem egyezik meg a Password mezőbe írt értékkel.
Invalid token.	Egy már felhasznált linkkel próbált újra jelszót változtatni.

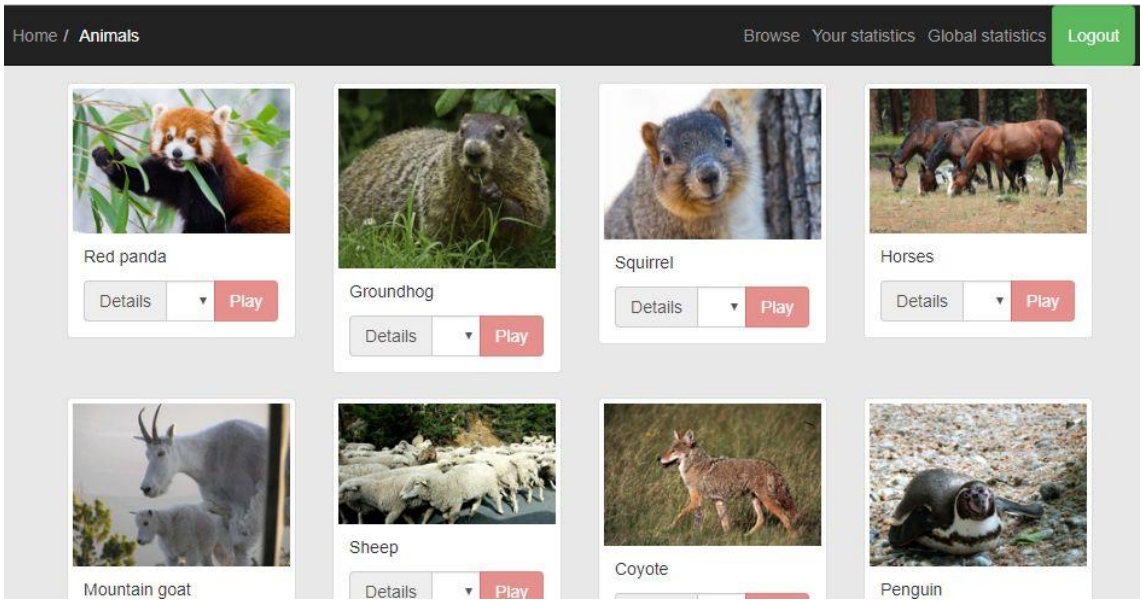
2.5 Kizárás

Abban az esetben, ha egy felhasználó ötször rosszul próbálkozik a belépéssel, akkor az adott felhasználót 5 percre kizárja a rendszer, amiről a kizárás oldal megjelenítésével tájékoztatja.

2.6 Böngészés játékosként

Képeket és kategóriákat böngészni csak bejelentkezett játékosok tudnak. A bejelentkezés után a rögtön a kategória böngésző oldalra jutunk, ami elérhető a `Home,` illetve a `Browse` linkre kattintással is. A kategória-böngésző oldalon a kategóriák jelennek meg és a

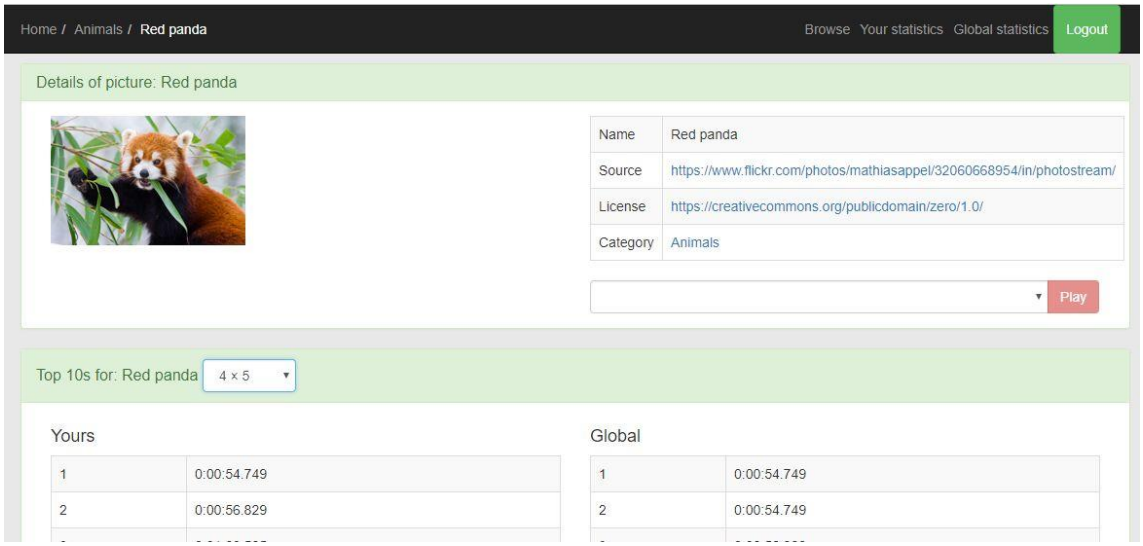
kategóriára kattintva jutunk el a képböngésző oldalra, ahol az adott kategória képei vannak felsorolva. Minden kép alatt megtalálhatóak a `Details` és `Play` gombok, valamint felvágás-választó legördülő menü. Itt már rögtön el lehet indítani egy játékot, ha választunk egy felvágást és a 'Play' gombra kattintunk.



2. ábra Böngészés egy kategóriában

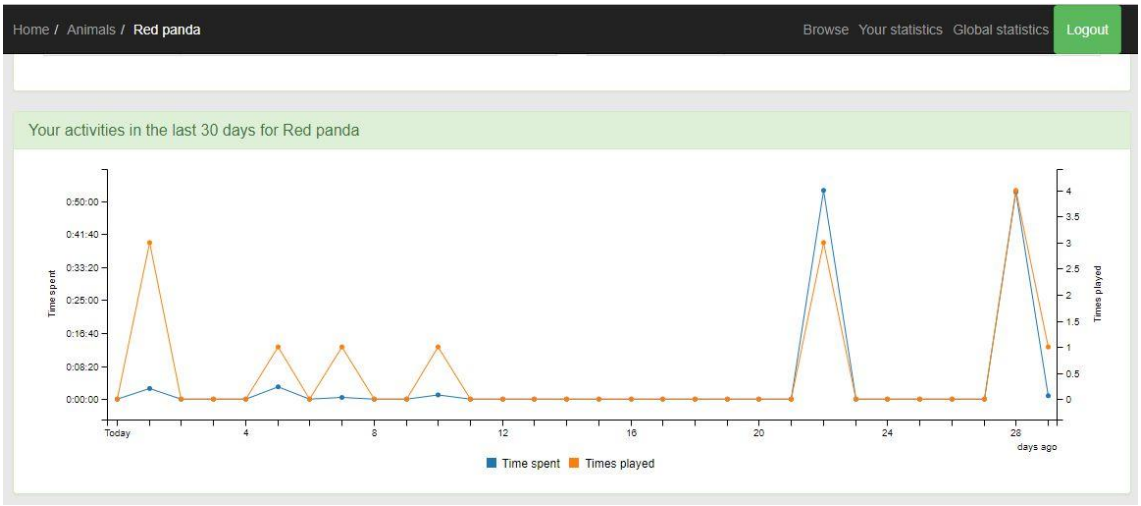
2.7 Információk a képről oldal

Erre az oldalra a képek alatti `Details` gomb megnyomásával juthatunk. Ezen az oldalon tudjuk megnézni a képhez kapcsolódó információkat. A kép nevét, a forrást, ahonnan a kép származik, a képhez tartozó licence információt, ami kép eredeti forrásától származik és a kép kategóriájának a nevét.



3. ábra Képinformációk és legjobb idők

Itt kapott helyet a képhez tartozó legjobb idők megjelenítése is a `Top 10s for: <kép név>` bekezdésben. A legördülő listából ki kell választani a felvágást, amihez meg szeretnénk nézni a legjobb időket. Ezután röviddel megjelennek legjobb idők: baloldalon a mi legjobb időink, a jobb oldalon pedig az abszolút legjobb idők.
A legjobb idők rész után jelenik meg a grafikon a mi aktivitásunkról az utolsó 30 napban (`Your activities for the last 30 days for <kép név>`). Kék színnel jelenik meg az, hogy hányszor játszottunk az adott képpel (`Times played`), narancssárga színnel pedig, hogy mennyi időt töltöttünk az adott képpel játszva (`Time spent`). A jelmagyarázat elemeire kattintva ki és bekapcsolhatjuk az adott elemhez kapcsolódó grafikont.



4. ábra Aktivitás az utolsó 30 napban

Ezalatt található az összes felhasználó aktivitása az utolsó 30 napban (`Global activities for the last 30 days for <kép név>`), ez funkcionalitásban teljesen megegyezik az előző grafikonnal, csak a megjelenített adatok mások.

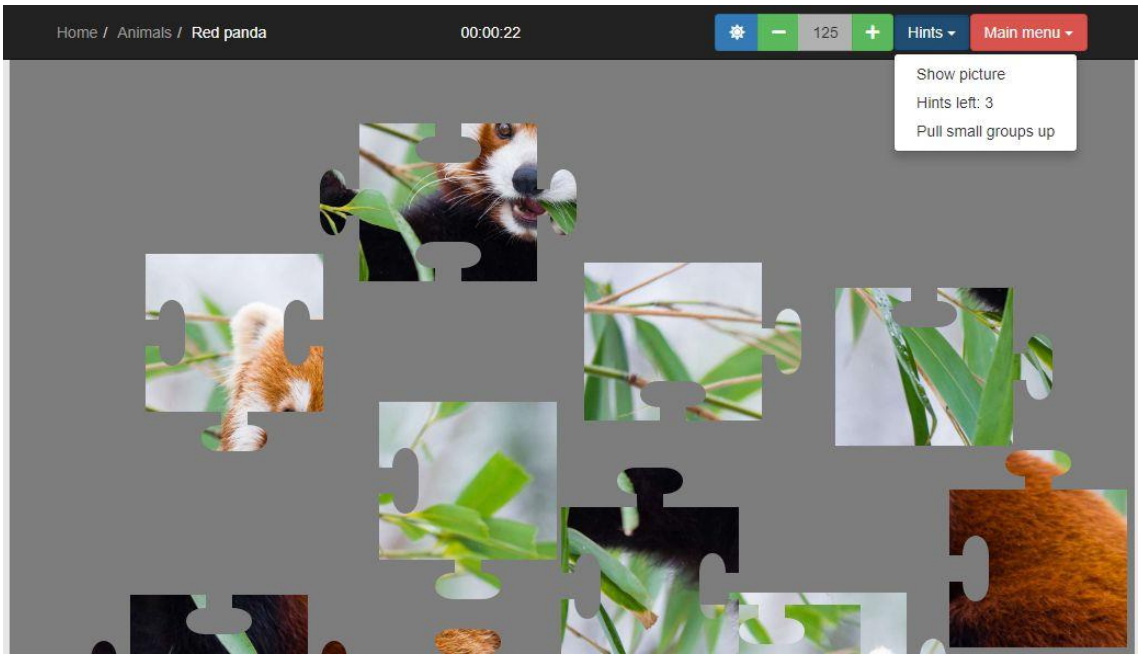
2.8 A játék

A játék a felvágás kiválasztása után elindítható a képböngésző, illetve az információk a képről oldalról is.

2.8.1 Navigációs sáv változások

A játék indításakor megváltozik a navigációs sáv. A korábban itt elérhető funkcionalitások a főmenü (`Main menu`) gomb alá kerülnek. A sáv közepén megjelenik

a játékban eltelt idő kijelzése. A főmenü gombtól balra láthatóvá válik a segítségek (`Hints`) gomb, mellette pedig a háttér világosságát módosító gombok.



5. ábra Játék

2.8.2 Segítségek és háttér fényesség

Játék közben van lehetőségünk háromféle segítséget kérni a kirakáshoz.

Megjeleníthetjük a képet összerakva egy modális ablakban a mutasd a képet linkre kattintással (`Show picture`).

Kérhetjük, hogy mutasson egy lehetséges összekapcsolást a segítség linkre kattintással (`Hints left: <a még kérhető segítségek száma>`), amit három alkalommal tudunk megtenni egy játék alatt. Amikor erre a linkre kattintunk, akkor fehér körök jelennek meg azokon az elemeken, amelyek összekapcsolhatók.

Előfordulhat, hogy nagyobb összekapcsolt csoportok eltakarnak egyes elemeket, amelyek így nem láthatók. Ebben ad segítséget a kisebb elemeket fentre húzó link (`Pull small groups up`), amire kattintva újrarendezi az elemeket, hogy a legkisebb csoportok kerüljenek legfelülre.

Azokban az esetekben, amikor egyes játék elemek nehezen láthatók, lehetőség van állítani a háttér fényességén. Ezeket a gombokat a mellettük található nap ikon azonosítja és a plusz/mínusz gombokkal állítható a fényesség. Ez a beállítás megmarad játékok között.

2.8.3 A játék elemek mozgatása és összekapcsolások

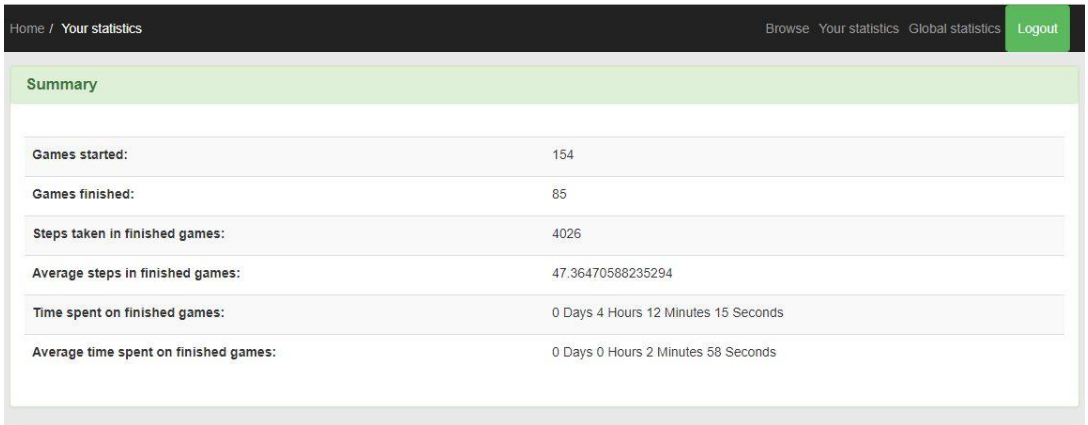
Játék közben az elemeket rákattintva és húzva tudjuk mozgatni, illetve a már összekapcsolt elemeket egy csoportként lehet mozgatni. Kattintás és húzás hatására az elem legfelülre kerül. Az elem elengedése után az algoritmus kiértékeli, hogy elég közel és megfelelő pozícióban van-e másik elemekhez, amelyekkel összekapcsolható. Ha ez igaz, akkor összekapcsolja az elemeket. Abban az esetben, ha a húzott elem kikerülne a játéktérről, akkor elengedés után visszahúzza egy olyan pontra, ahol már teljesen a játéktéren van.

2.8.4 A játék vége

A játéknak akkor van vége, amikor minden elemet összekapcsoltunk, és ezt egy modális ablak megjelenése jelzi. Itt láthatjuk a képet kirakott formában, valamint, azt, hogy mennyi idő alatt sikerült kirakni azt. Az ablakot az alján található két gombbal lehet eltüntetni. Az új játék (`Play again`) gombbal újra játszhatunk ugyanazzal a képpel. A vissza a böngészéshez (`Back to browse`) gombbal pedig a kategória-böngésző oldalra mehetünk.
Azok a statisztikák, amelyek csak a játékosra vonatkoznak, azonnal frissülnek, amelyek minden játékosra vonatkoznak, azok néhány perccel később.

2.9 A játékos statisztikái

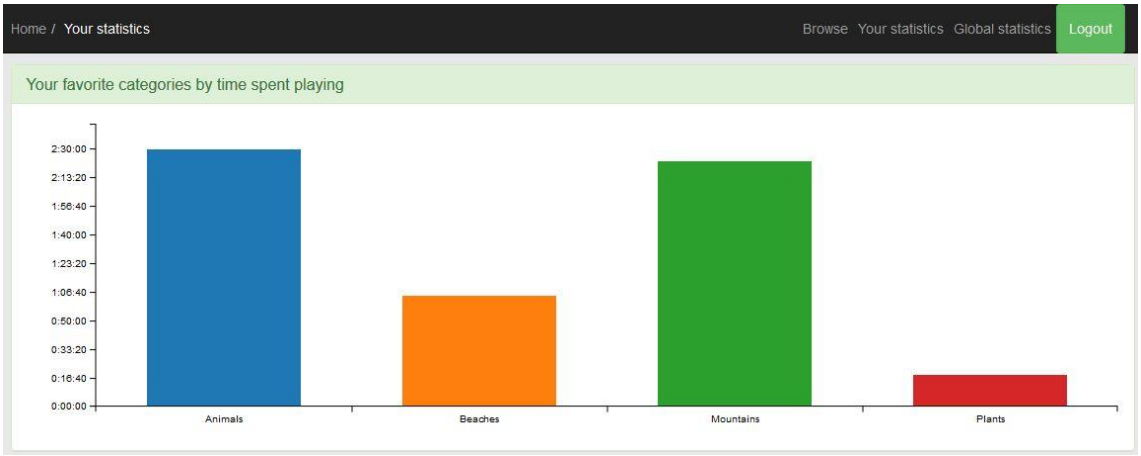
A játékosra vonatkozó statisztikák a te statisztikáid (`Your statistics`) menüpont alatt érhetők el a navigációs menüből.



6. ábra Játékos statisztikái összegzés

Az első rész az Összegzés (`Summary`), ahol megjelennek egymás alatt a főbb adatok az eddigi tevékenységünkről:

- az elkezdett játékok száma (`Games started`)
- a befejezett játékok száma (`Games finished`)
- befejezett játékok lépésszáma (`Steps taken in finished games`)
- befejezett játékok lépésszám átlaga (`Average steps in finished games`)
- befejezett játékok összesített ideje (`Time spent on finished games`)
- befejezett játékok átlagideje (`Average time spent on finished games`)



7. ábra Játékos statisztikái kedvencek

Az ezután következő rész azt mutatja meg, melyik a kedvenc kategóriánk az alapján, hogy mennyi időt töltöttünk el játékkal, illetve hányszor játszottunk képekkel az adott kategóriából (`Your favorite categories by time spent/times played`). Az itt megjelenített adatok csak azokra a játékokra vonatkoznak, amelyeket sikeresen fejeztek be a felhasználók, és legfeljebb 10 kategóriáról mutatok meg adatokat.
Az utolsó részben szintén ezek az adatok jelennek meg, de ebben az esetben már a képekre vonatkozóan.

2.10 Mindenki statisztikái

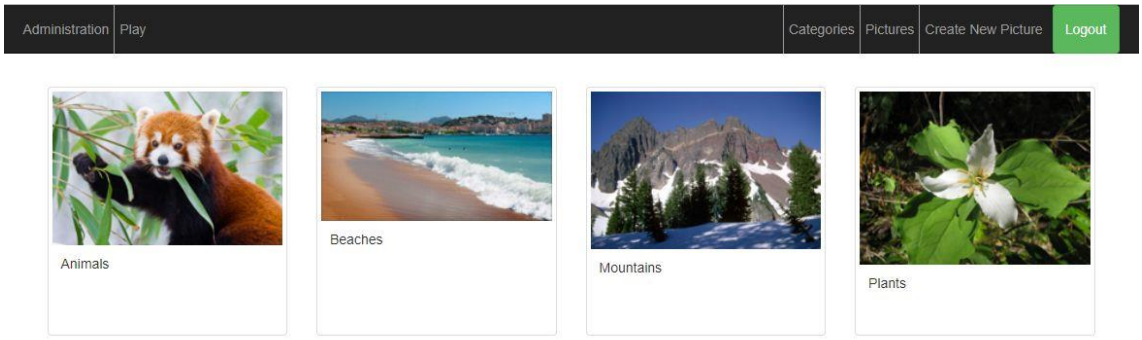
A minden játékos statisztikái (`Global statistics`) ugyanazokat a típusú elemeket jelenítik meg, mint a játékos statisztikái, azonban itt az összes játékosra vonatkoznak. Ezek az adatok egy játék után nem azonnal frissülnek, hanem néhány perccel később.

2.11 Adminisztrációs felület

Az adminisztrációs felületet csak azok a felhasználók érik el, akik az ehhez megfelelő Alkalmazás Adminisztrátor (`Admin`) szerepkört megkapták. Az ilyen felhasználók belépés után rögtön erre a felületre vannak irányítva, ha játszani szeretnének, akkor innen kell tovább lépniük a játékos részekre.

2.12 Adminisztrátor navigációs sáv

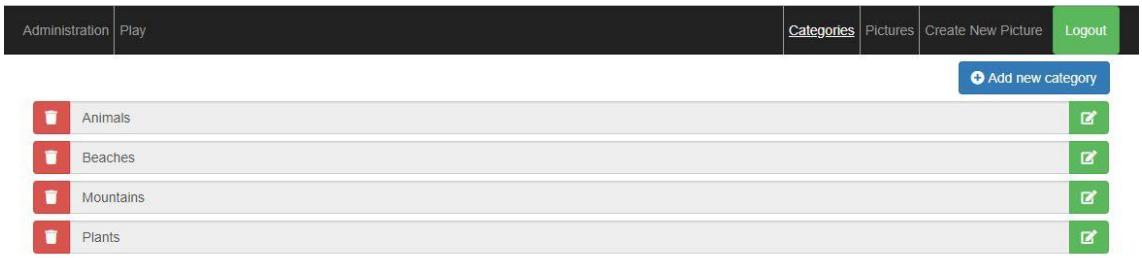
Az adminisztrátorok navigációs sávján először az Adminisztráció (`Administration`), illetve Játék (`Play`) linkeket láthatjuk. Az Adminisztráció link az adminisztrációs felület fő oldalára vezet, míg a Játék link pedig átvezet a játékosok főoldalára.
A többi funkció: a kategóriák listája (`Categories`) és kapcsolódó funkciók, a képek listája (`Pictures`) és kapcsolódó funkciók, új kép létrehozása (`Create new picture`) valamint a kilépés (`Logout`).



8. ábra Kategória lista adminisztrátoroknak

2.13 Kategóriák listája adminisztrátoroknak

Itt láthatjuk a már létező kategóriákat, módosíthatjuk a nevüket és hozhatunk létre újabbat.



9. ábra Kategória módosítás lista

A kategória lista minden eleménél a jobb oldalon található gombbal lehet törölni egy kategóriát, és a baloldalon található gombbal tudjuk módosítani a kategória nevét. Egy kategória csak abban az esetben törölhető, ha már nincsen hozzátartozó kép.
Egy kategória létrehozásához az Új kategória hozzáadása (`Create new category`) gombra kell kattintani, majd név kitöltése után, a Létrehozás (`Create`) gombra kattintva tudjuk elmenteni. Egy új kategória nem jelenik meg a Játékosoknak egészen addig, amíg nincs hozzá feltöltve legalább egy kép.

2.13.1 Hibaüzenetek a kategóriák módosításánál

10. ábra Használt kategória hibaüzenet

Hibaüzenet	**Hiba oka**
The category cannot be deleted it is in use.	Olyan kategóriát próbált törölni, amelyhez van kapcsolódó kép
There is already an entry with the name <kategória név>	Olyan nevű kategóriát próbált létrehozni, amely már létezik.
Name is required	Nem adott meg kategória nevet.
Name cannot be longer than 100 characters.	100 karakternél hosszabb nevet adott meg.

2.14 Képek listája adminisztrátoroknak

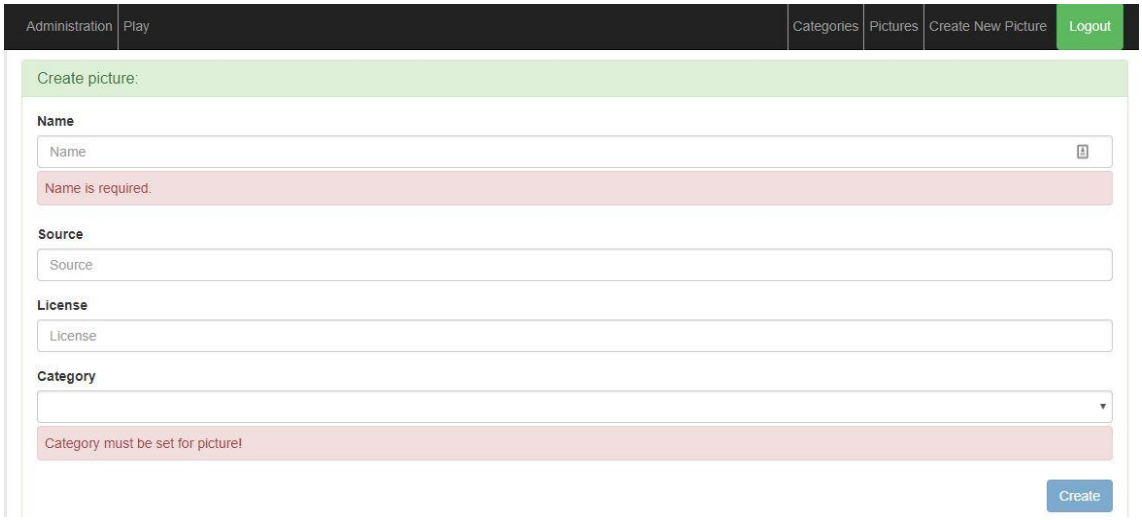
Adminisztrátori belépés után, rögtön a kategóriákra osztott képek listáját látjuk, amelyből egy kategóriát kiválasztva jutunk el az adott kategóriához tartozó képekhez. Az így megjelenő oldalon listaként vannak felsorolva a képek. Minden elemhez megjelenik egy kis méretű kép, mellette a főbb információk, illetve a lehetőség egy kép törlésére, valamint adatainak módosítására.

2.15 Új képadat létrehozása

Új képadatot az új kép létrehozása linkre (`Create New Picture`) kattintva tudunk létrehozni. Itt lehet megadni a kép fő adatait: név (`Name`), honnan származik a kép (`Source`), az eredeti licenszt (`Licence`), ami a kép forrásától származik és a kép kategóriáját (`Category`).
Mindenképpen meg kell adnunk egy nevet és egy kategóriát a képadathoz, ezután tudjuk elmenteni a Létrehozás (`Create`) gombra kattintva. A sikeres mentés után a kép módosítása mező jelenik meg, ahol fel tudunk tölteni képet a képadatunkhoz.

2.15.1 Hibaüzenetek új képadat létrehozásánál

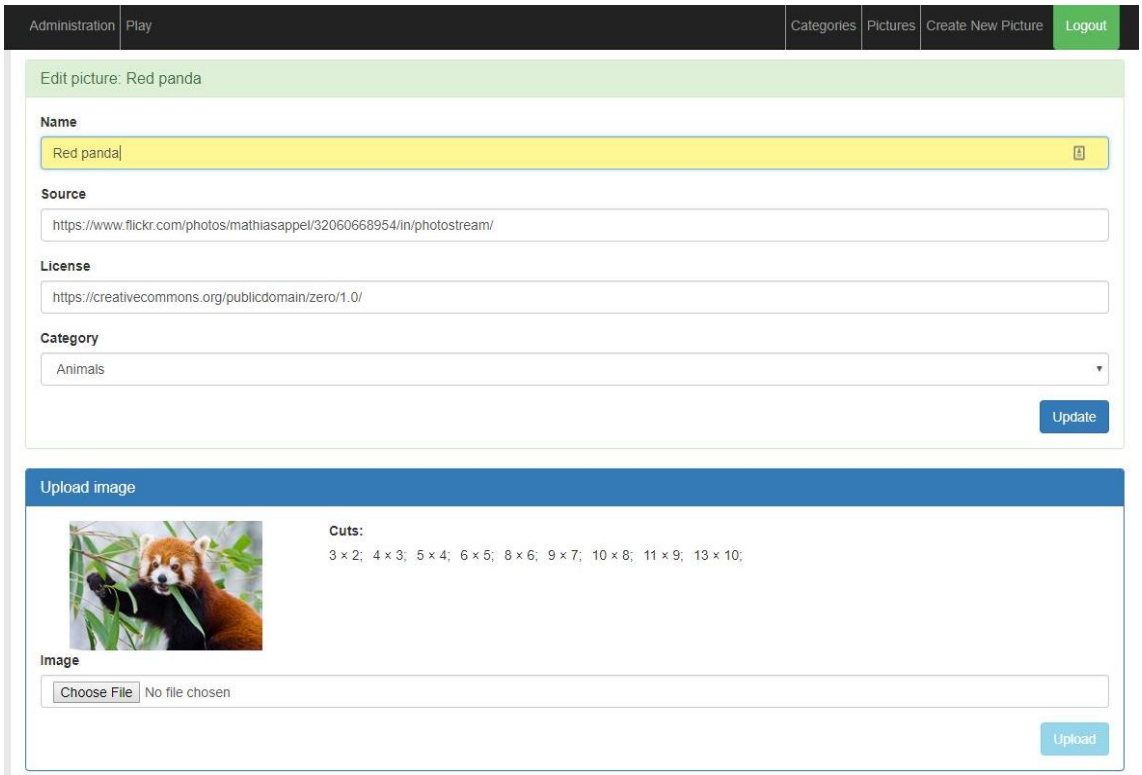
Hibaüzenet	Hibaüzenet oka
Name is required	Nem adott meg nevet
There is already an entry with the name <kép név>	Az adott névvel már létezik kép
Name cannot be longer than 100 characters	100 karakternél hosszabb nevet adott meg
Source cannot be longer than 1000 characters	Forrás megjelölésnek 1000 karakternél hosszabb szöveget adott meg
Licence cannot be longer than 1000 characters	Licensznek 1000 karakternél hosszabb szöveget adott meg



11. ábra Képadat hibaüzenetek

2.16 Kép módosítása

A kép módosítása oldalra a képek listájáról juthatunk. Itt megváltoztathatjuk a kép nevét (`Name`), a kép forrását (`Source`), az eredeti licenszt (`Licence`, és a kép kategóriáját is (`Category`). A változtatások után a Mentés (`Update`) gombra kattintva mentünk. A kép fő adatait felsoroló rész alatt találjuk a kép feltöltésének lehetőségét és ha már van feltöltve kép, akkor itt látható az előkép és a lehetséges felvágások.

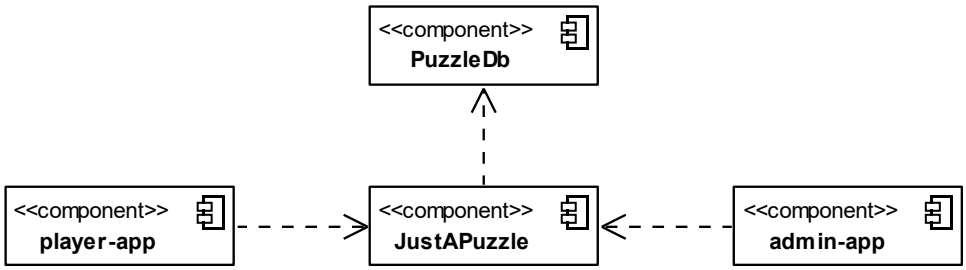


12. ábra Kép módosítás

3 Fejlesztői dokumentáció

3.1 Az alkalmazás tervezett felépítése

Az alkalmazás egy webalkalmazás, ebből adódóan szüksége lesz egy szerver alkalmazásra, ami ki tudja szolgálni a felhasználóktól érkező kéréseket. A tervezett funkcionalitások alapján két szerepkör adódik: a Játékos, illetve az Adminisztrátor. Két külön TypeScript alkalmazást hozok létre annak érdekében, hogy ne a böngészőben futó alkalmazásnak kelljen eldönteni, melyik felhasználó mihez férhet hozzá. Ezt kiegészítem szerver oldali ellenőrzéssel is, mert az alkalmazás végpontjai ettől függetlenül elérhetőek lesznek mindenkinek. Minden adatot adatbázisban fogok tárolni.
Így alakul ki a struktúra, ami lejjebb látható a szerver neve (`JustAPuzzle`), a játékosok alkalmazása (`player-app`), az adminisztrátoroké (`admin-app`) és az adatbázis (`PuzzleDb`).



13. ábra Az alkalmazás komponensei

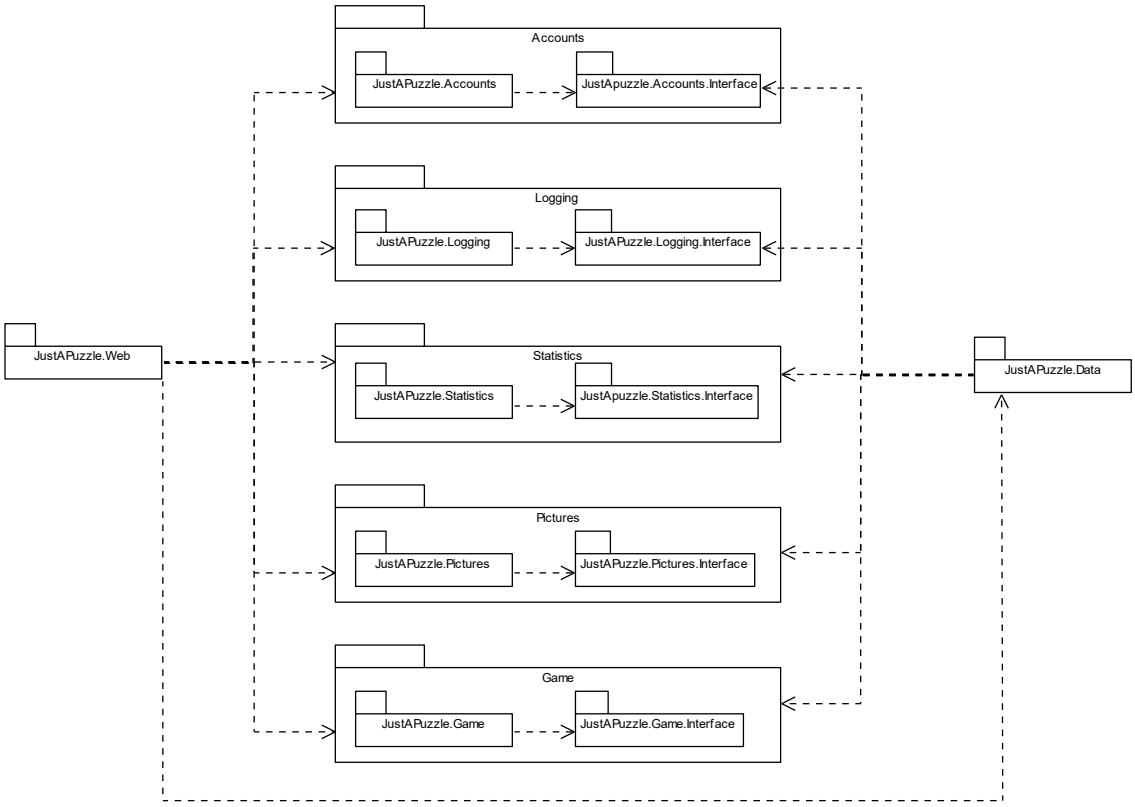
Ezt a struktúrát majd a forráskód is meg fogja jeleníteni. A kliensek kódját a `clients` mappába, a szerver kódját a `server` mappába fogom elhelyezni.

3.2 Szerver alkalmazás terve

A szerver alkalmazásnak az ASP.NET Core megoldás adja az alapját felhasználva, az ehhez kapcsolódó ASP.NET Core Identity és Entity Framework Core csomagokat a felhasználók adatainak, illetve az adatbázis kezelésére.
Szerver oldalon alkalmazom a már szokásosnak mondható megjelenítés, logika és adat réteg felbontást. Az alkalmazás réteget tovább bontom a felhasználási esetek szerint, a többi réteg esetében az elvégzendő feladatok mérete nem akkora, ami egy ekkora

szétválasztást indokolna. Ha később erre szükség lenne, akkor az alkalmazás réteg csomagjai már előre vetítik a lehetséges felbontások további formáját.

Az alkalmazás rétegben a komponensek implementáció és interfész részre vannak szétválasztva elősegítendő a tesztek írását és a gondolkodási módot, aminél nem az implementációt veszem figyelembe ezek felhasználásánál. Az interfész csomag definiálja az implementációs osztályok publikus interfészét, és a kivételeket, amelyeket a csomagban található osztályok dobhatnak. Emellett ez fogja tartalmazni a szükséges függőségek interfészeit, amelyeket más csomagokban fogok implementálni, ilyenek például az adatíró, illetve olvasó osztályok. Az interfész azokkal az adatosztályokkal lesz teljes, amelyeket a csomagban használt metódusok várnak paraméterként vagy adnak vissza értékként.

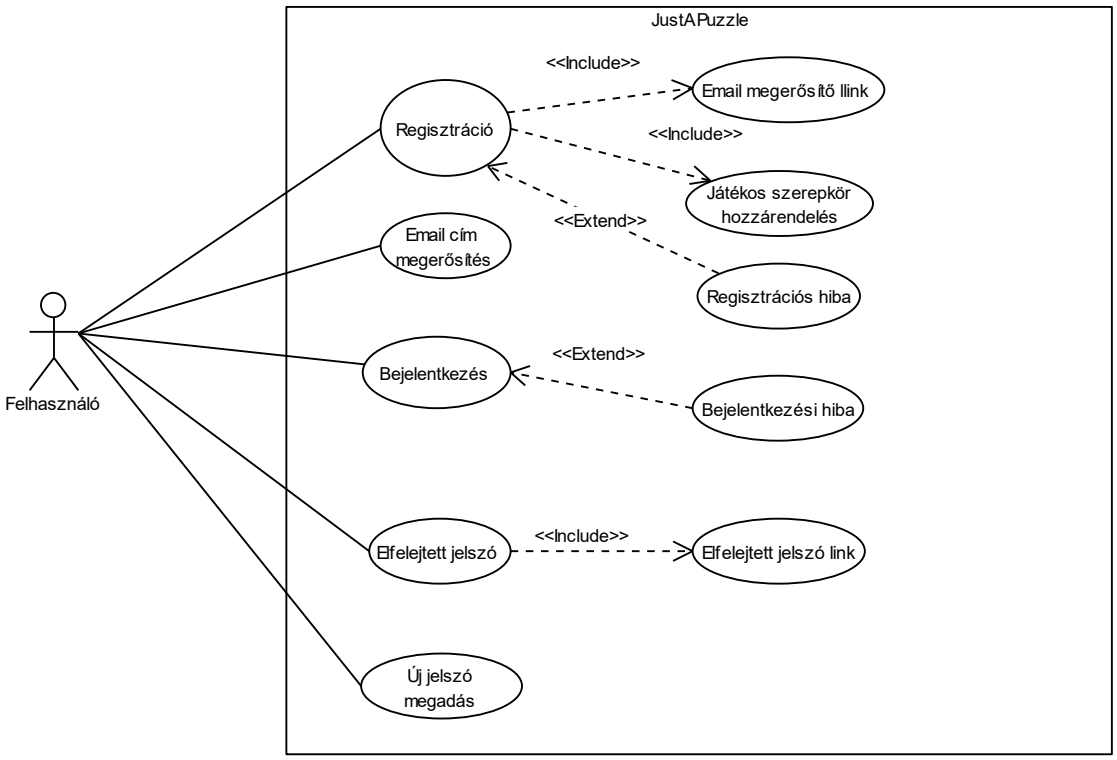


14. ábra Szerver csomagok

A felhasználó csak a felhasználói adatok kezelése esetén fogja közvetlenül használni a szerver alkalmazást, ezeket az esetek jellemzi a következő diagram. Minden más esetben a kliens alkalmazások fognak kommunikálni a szerverrel.

Mindkét felhasználó típus esetén a felhasználót a megfelelő kliens alkalmazás belépési pontjára irányítom, és amint bejelentkezett, az így kapott html-ek fogják referenciálni

kliens alkalmazások fájljait. Játékosok számára ez a `HomeController Index` metódusa, illetve az adminisztrátorok esetén az `AdminController Index` metódusa.



15. ábra Szerver közvetlen felhasználói eset diagram

3.2.1 Hibakezelés

A kliens oldalon ellenőrzöm, hogy helyes információk kerüljenek a szerver alkalmazáshoz, de hibák ettől még előfordulhatnak, illetve az ügyesebb felhasználók meg tudják kerülni az ellenőrzéseket. A szerver oldalon ezért kezelni kell a hibákat és menteni kell az információkat róluk. A mentéshez lehet ugyan választani, hogy minden hibaellenőrzéshez beinjektálok egy függőséget, ami ezt megteszi, de ezzel bonyolítanám az alkalmazás logikáját a működéshez szükségtelen lépésekkel. Ezért azt választottam, hogy minden hibánál, amit az alkalmazás réteg osztályai észrevesznek, egy megfelelő kivételt dobok, amelyet a megjelenítési rétegben kezelek. Ez még mindig csak a probléma áthelyezését jelenti, de szerencsére a keretrendszer erre ad megoldást, amit felhasználhatok, ezek a kivétel kezelő attribútumok. Ezeket az attribútumokat felteszem minden olyan kérés kezelő metódusra, amelynél az adott kivétel előfordulhat.
A kivételkezelő attribútumok fogják elvégezni a mentést, illetve létrehozni a megfelelő választ a felhasználónak.

A következő hibákat kezelem ilyen módon:

- Email cím megerősítésnél nem regisztrált felhasználó, nem megfelelő link
- Belépésnél nem megfelelő felhasználó név és jelszó pár, kizárt felhasználó
- Jelszóváltoztatásnál nem létező felhasználó, illetve bármilyen ezzel kapcsolatos egyéb hiba
- Adott névvel már létező kategória vagy kép létrehozására történő próbálkozás
- Használt kategória törlése
- Olyan adatot küld a kliens alkalmazás egy játékról, ami a játék adott állapotában nem lehetséges
- Nem létező felvágással próbál játékot indítani a felhasználó
- Nem létező képpel próbál játékot indítani a felhasználó

A felhasználótól kapott adatok formáját is lehet ellenőrizni, ezek már részei a keret rendszernek. Mivel az ilyen hibák nem fordulhatnak elő normál felhasználásnál, ezért ezeket is menteni szeretném, és erre is megoldás egy megfelelő attribútum implementálása.
Az ilyen attribútumok a megjelenítési csomag `AccountFilters` és `Filters` könyvtáraiban találhatók meg.

3.3 Megjelenítési réteg

A megjelenítési réteg, a már rég óta létező ASP.NET MVC mintát követi. A böngészőtől érkező kérések url-je alapján választ a rendszer egy Controller osztályt, annak egy metódusát és a paraméterként adott adatokat behelyettesíti a metódus paramétereként. Attól függően, hogy a html-t vagy adatot vár, a kérés ad adat vagy html választ.
A megjelenítési réteg egyben az alkalmazás belépési pontja is, itt találhatóak meg regisztráció és egyéb felhasználói adatkezeléshez szükséges megjelenítési fájlok, valamint a kliens alkalmazások fájljai is ide kerülnek a fordítás után.
A függőségek regisztrációja is ebben a rétegben történik. Itt adom meg azt, hogy a függőségként kezelt interfészekhez mely konkrét implementációk tartoznak, ez a `Startup` osztályban történik.
A választott felhasználó kezelési csomag (ASP.NET Identity) megköveteli a fő megjelentési csomagok hozzáadását is, emiatt csomagoló osztályokat hoztam létre

(`SignInManagerWrapper, UserManagerWrapper`), így a felhasználók kezelését végző alkalmazás osztályoknak nincs szükségük ezekre. Ennek hatására a felhasználókat kezelő osztályokhoz egység teszteket tudok írni hosszas konfiguráció nélkül.
Két előre nem látott implementációs problémát is kellett kezelni ebben a rétegben. Az első és a fájóbb probléma, hogy az EntityFramework migrációs osztályokat csak akkor tud létrehozni, ha az adatbázist modellező Context osztályt csak a belépési pont szerepet ellátó csomagokban helyezem el. Ezért ide kerültek a migrációk és a modellező osztály (`ApplicationDbContext`), annak ellenére, hogy ezeket az adatrétegben szerettem volna elhelyezni. A másik felmerült probléma, hogy az email cím megerősítéshez, illetve jelszóváltoztatáshoz szükséges linkeket is csak itt lehet generálni, ezért létrehoztam egy függvény szignatúrát (delegate) az alkalmazás rétegben, amihez a megfelelő függvényt a megjelenítési réteg hoz létre, majd ad át az alkalmazás rétegnek.

3.4 Alkalmazás réteg

3.4.1 Felhasználók kezelése

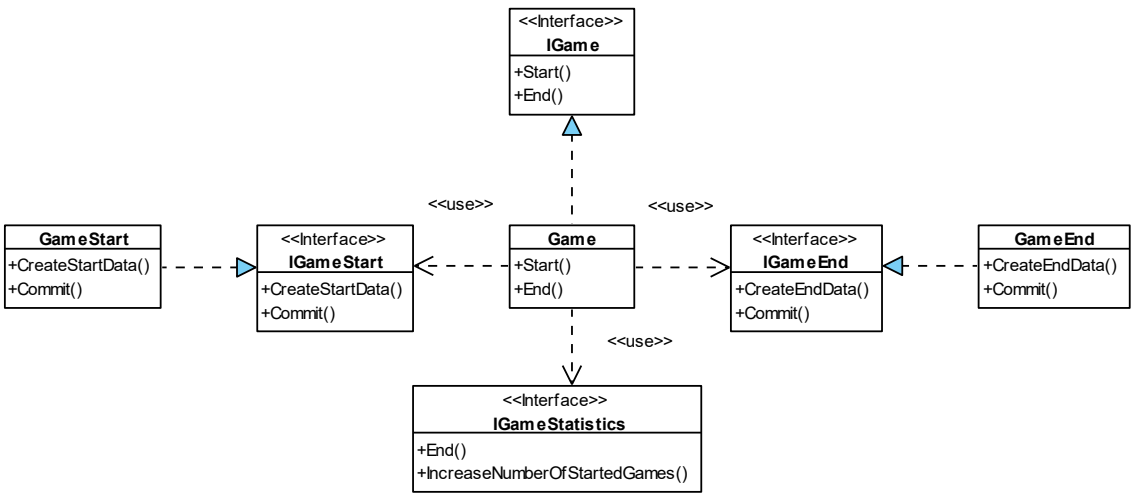
Az alkalmazás felhasználóinak szüksége van regisztrációra, elfelejtett jelszó visszaállításra, kilépésre és belépésre. Az ezzel kapcsolatos funkcionalitások nagy része már rendelkezésre áll a keretrendszerben. A használatukhoz ugyan nem kellene létrehozni külön modult, hiszen mehetne minden egyetlen `Controller`-be, ahogy ezt egy új ASP.NET Core MVC alkalmazás létrehozásakor kapom, viszont ebben az esetben nagyon nehéz tesztekkel kifejezni, hogy milyen működést várok el egy felhasználói fióktól. Emiatt inkább azt választottam, hogy csomagoló osztályokkal megjelenítem a várt funkcionalitásokat. Ilyen átalakítás után el tudom érni, hogy visszaadott hibaértékek helyett kivételek jelezzék a hiba megtörténtét.
A felhasználó fiókot az `Account` osztály jeleníti meg, implementálva a fő funkcionalitásokat: belépés, kilépés, létrehozás, emailcím megerősítés, jelszó helyreállítás, valamint annak eldöntése, hogy a felhasználó adminisztrátor-e.
Szükség van egy email küldő funkcionalitásra is, ami kezeli az email megerősítés és elfelejtett jelszó emailek küldését. Ez áll a kétfajta emailt megvalósító osztályból, amelyek elvégzik az üzenetek összeállítását és egy osztályból, ami elküldi őket.

3.4.2 Játék adatok kezelése

A játékot magát egy külön osztály jeleníti majd meg (`Game`) és ennek feladata nem lesz más, mint összefogni a játék kezdet és játék befejezés funkcióit a statisztikai adatok készítésével. Ebből kapom a következő struktúrát: egy játék kezdet osztály (`GameStart`), ami csak a játék kezdetet valósítja meg; egy játék vége osztályt (`GameEnd`), ami csak játék végét valósítja meg; és végül egy játék statisztika osztályt (`GameStatistics`), ami a statisztikai adatok frissítését hajtja végre. A `GameStatistics` osztályt a statisztikai csomagban implementálom.

Egy játék kezdetén ellenőrzöm, hogy létezik-e a kép, amivel játszunk, illetve a megadott felvágás létezik-e az adott képhez. Amennyiben ezek léteznek, akkor elmentem a játék kezdet adatokat (felhasználó, kép, felvágás, kezdés ideje) és generálok egy játék azonosítót. Játék végén ellenőrzöm az alábbiakat: létezik-e a játék, a státusza nem befejezett-e, a megfelelő játékos adatai vannak-e társítva hozzá, a játék kezdete korábbi időpont-e, mint a vége, illetve a játék végén minden elem össze van-e kapcsolva. Ezután elmentem, hogy mikor végződött a játék és a megtett lépéseket.

A játék osztály kezdő metódusa (`Start`) fogja össze a játék adatbeszúrást a kezdő statisztikai adatokkal egyetlen tranzakcióba, ugyanez történik a vége metódusban is (`End`) csak a vége adatokkal.



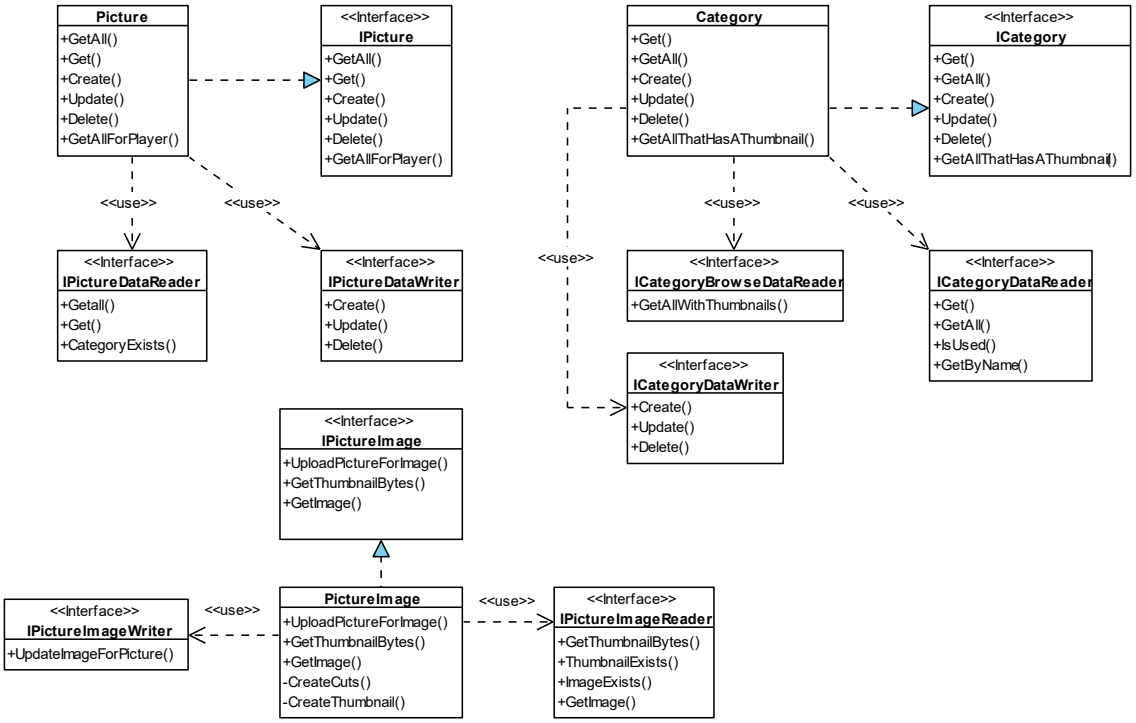
16. ábra Game csomag osztályai

Implementáció során merült fel, ha teszteket szeretnék írni, akkor szükség lesz még két osztályra: egy azonosító generáláshoz, egyre pedig az idő meghatározásához. Ezek csak csomagoló osztályok a C# nyelvben már létező megoldásokhoz, amelyek nem

implementálnak interfészeket, ezért az általuk visszaadott értékek nem mockolhatók a teszthez.

3.4.3 Kép és kategória adatok kezelése

Ebben komponensben is főosztályok implementálják a funkcionalitásokat. A kategóriák kezelését a `Category` osztály végzi, ennek részei minden kategória olvasása, itt meg kell különböztetni, hogy játékos vagy adminisztrátor böngészi a kategóriákat. Első esetben csak azok a kategóriák jelennek meg, amelyekhez már van legalább egy képadat, amihez már töltöttek fel képet. A második esetben minden kategória megjelenik. Egy adott kategóriához mindig az első hozzátartozó kép előnézetét adja vissza az alkalmazás. Kategória létrehozásnál és frissítésnél mindig ellenőrizni kell, hogy van-e már ilyen nevű és ha már van, akkor megfelelő kivételt dob a metódus. Kategória törlése esetén szintén lefut egy ellenőrzés, hogy a kategóriához van-e kép, ha van, akkor ezt megfelelő kivétel dobásával jelezzük.
A képadatokat a `Picture` osztály kezeli, létrehozásnál és frissítésnél figyelem, hogy van-e már ilyen nevű kép, illetve létezik-e a kiválasztott kategória, bármelyik hibánál kivétellel jelzem a hibát.

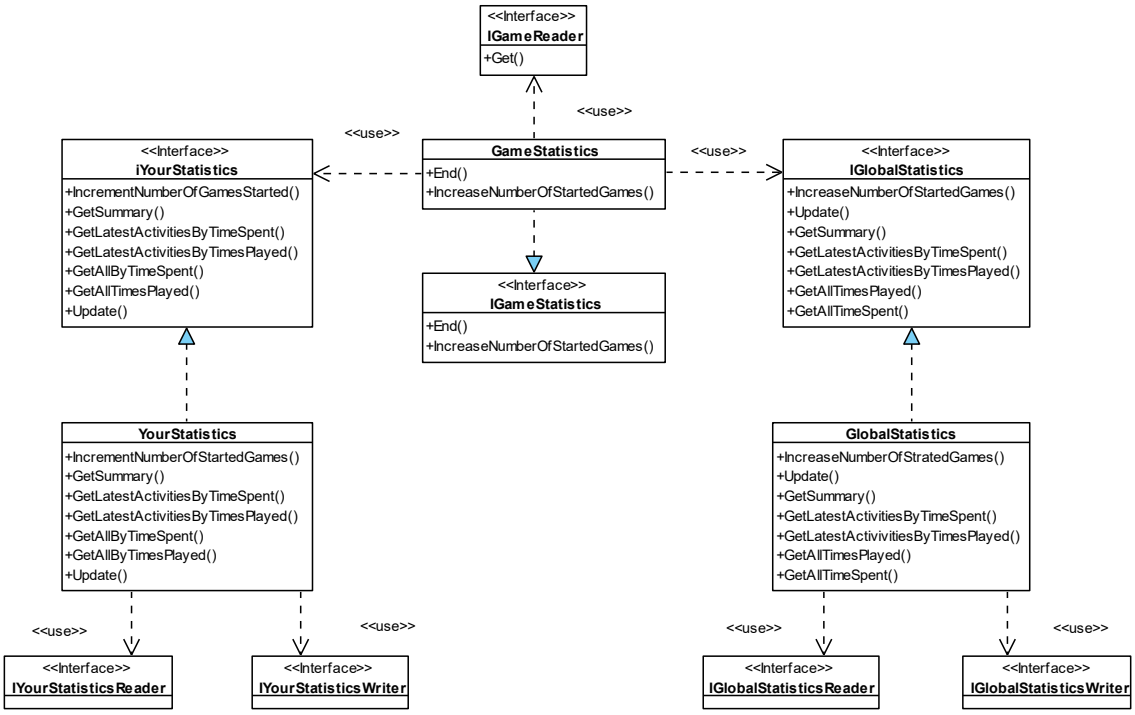


17. ábra Picture csomag osztályai

A képadathoz tartozó képek mentését és visszaadását a `PictureImage` osztály végzi. Képfeltöltésnél először létrehozom az előképet, ami megjelenik képböngészésnél, illetve, ha ez az első kép a kategóriában, akkor kategória-böngészésnél. Ezután következik a felvágások meghatározása: minden felvágásnál az a cél, hogy a kapott kisebb darabok szélesség, illetve hosszúság aránya 6:5-höz legyen. A képeket mindig adott mennyiségű sorra vágom fel és ehhez határozom meg az oszlopok számát. Először veszem a részmagasság hatszorosát, majd osztom öttel. A kép teljes szélességét osztva az így kapott értékkel jutok el az alsó oszlop értékhez, amihez adva kapom meg a felső oszlop értéket. A két értékhez meghatározom, hogy mekkora teljes képszélességet generálna, majd azt választom, amelyiknél kisebb az eltérés az eredeti szélességhez képest.
Az implementáció során merült fel, hogy érdemes lecserélni az eredeti fájl nevet egy egyedi azonosítóra, mivel így elejét tudom venni, hogy ugyanolyan nevű képek tartozzanak két különböző kép adathoz. Továbbá, mivel a képeket később majd fájlnév alapján érem el és abban az esetben, ha valaki újabb képet töltene fel ugyanazzal a fájlnévvel, akkor letöltésnél a böngésző ezeket cache-elné, emiatt a változás nem minden felhasználóhoz jutna el.

3.4.4 Statisztikák

Az alkalmazás által vezetett statisztikáknak két különböző fő kategóriája van a játékos és az összesített statisztikák. Mindkét kategória alatt megtalálhatóak a következők: a legjobb eredmények egy képhez felvágás szerint; egy összegző statisztika; kedvenc kategóriák és képek az eltöltött játékidő, illetve a befejezett játékok száma szerint; egyes képekhez az utolsó 30 nap adatai, eltöltött idő és befejezett játékok szerint.
A statisztikai adatok frissítését mindig a `GameStatistics` osztály indítja el, a játékos statisztikáinál ez mindig egyszerűen az adatbázis adatainak változtatása játék végén. Az összesített adatoknál azonban, ha véletlenül két játék egyidejűleg vagy közel egyidejűleg fejeződne be, akkor előfordulhatna, hogy elavult adatokat frissítenénk. Ezt elkerülendő játék indításakor és a végén egy frissítés kérő üzenetet mentek az adatbázisba, amelyeket később háttérfolyamatok dolgoznak fel kötegszerűen. A háttérfolyamatok osztályai kerültek ebbe a modulba, amelyek az `IHostedInterface`-t implementálva indítják ezeket a folyamatokat 5 perces időközönként. Az 5 perces limit most kódban van rögzítve, de könnyedén át lehetne alakítani, hogy ezt az alkalmazásban konfiguráljuk.



18. ábra Statistics csomag osztályai

3.5 Adat réteg

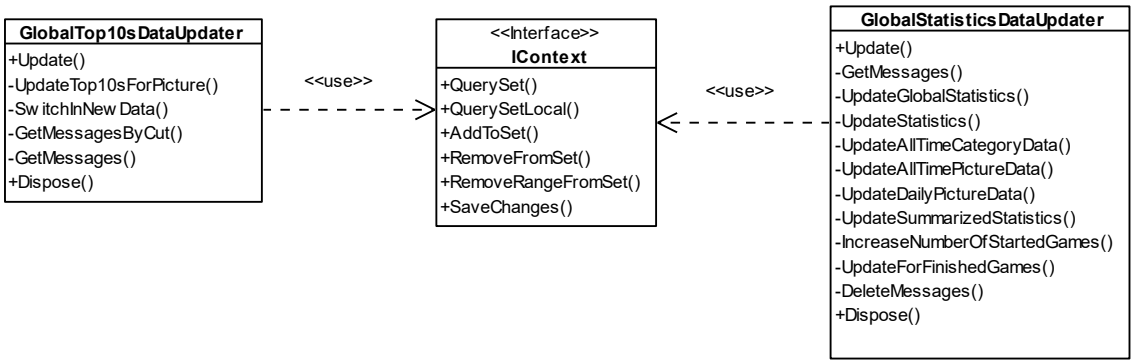
Az adatelérési rétegnél az EntityFramework Core csomagot használom és PostgreSql adatbázisba mentem az adatokat. Az EntityFramework használata miatt nem szükséges sql scripteket írni az adatbázis létrehozáshoz, csak generálni kell ezeket, ami mindig egy nagy segítség.

Az adatelérési rétegnél választhattam volna, hogy a nekem és sok más C# fejlesztőnek ismerős Repository mintát használom, de ezzel azonban több gond is akad. Amit általában Repository-ként lát egy fejlesztő, az csak egy csomagoló osztálya a `DbContext` osztálynak, ami az igazi Repository implementáció. Emiatt az elnevezés nem szerencsés, bár kifejezetten elterjedt. Az ilyen repository osztályok ráadásul szinte mindig erősen kötődnek az adatbázis tábláihoz, egy tábla, egy típus, egy repository. Emiatt az alkalmazások logikai rétegét gyakran meghatározza, az adatbázis felépítése, implementált funkcionalitások helyett. Az ilyen típusú kötődés magával hozza, hogy minden adatelérés egyetlen osztályba kerül, ebből következően ezek az osztályok mind nagyméretűek lesznek és ha több táblát kell frissíteni, akkor az alkalmazás réteg osztályai több repository-tól is függővé válnak. Ezért inkább író és olvasó osztályokat hoztam létre, amelyet a logikai réteg főbb osztályai használnak. Azért választottam szét a két funkcionalitást, mert tipikusan, ha olvasni szeretnénk az adatbázisból akkor, nincs

szükségünk az író metódusokra. Ráadásul, nem minden adat, amit olvasni szeretnénk kerül be úgy az adatbázisba először, mint ahogy azt olvasni szeretném.
Olyan adatra, ami nem úgy kerül be az adatbázisba először, mint ahogy azt később olvasni fogja az alkalmazás, jó példa az összesített adatok kezelése. Minden játék végén bekerül az adatbázisba egy frissítés üzenet (`GlobalUpdateMessage`), az összesített adatok frissítését elindítja a megfelelő statisztikai modulban lévő osztály (`GlobalStatisticsUpdater`) és frissítést elvégzi az összesített adatokat frissítő osztály (`GlobalStatisticsUpdater`).
A fentieken kívül itt találhatók még az adatbázis tábláknak megfeleltetett osztályok a hozzájuk tartozó konfigurációkkal, illetve a megjelenítési rétegnél már említett implementációs problémát feloldó `IContext` osztály. Amiatt, hogy az adatbázist jellemző `ApplicationDbContext` osztályt a megjelenítési rétegben kellett létrehozni, az adatrétegnek függenie kellett volna a megjelenítési rétegtől, mivel az író olvasó osztályok ezt használják. A függőségeket viszont injektáljuk, ezért egy megfelelő interfész létrehozása, annak implementálása és némi függőség regisztráció után ezt el tudtam kerülni.

3.5.1 Összesített statisztikák frissítése

Játék kezdetén az adatbázisba kerül egy üzenet, ami ez elkezdett játékok számának frissítését jelzi, a játék végén két frissítő üzenet kerül be: az első a legjobb eredmények frissítését jelzi, a másik pedig az összesített statisztikákét.
A játékok számának és az összesített adatoknak frissítését a `GlobalStatisticsDataUpdater` osztály végzi az alábbi módon. Először kiolvasom az üzeneteket, innen kezdve a később létrejövő adatokkal már nem foglalkozom. Ezután frissítem a grafikonokon megjelenített adatokat, majd az összegző adatokat. Miután mindent frissítettem az összes feldolgozott üzenetet törlöm az adatbázisból. Mivel ez egy háttérfolyamat, ezért itt nem alkalmazható az attribútumos hibakezelés, emiatt itt a folyamatot indító osztály végzi a hibák mentését.
A tíz legjobb eredmény mentése is háttérfolyamatként történik, mikor az összes játékosról van szó, ezt a `GlobalTop10SDataUpdater` osztály végzi el. A folyamat hasonló, mint az előző esetben, kiolvasom az üzeneteket, ezek alapján frissítem az adatokat, majd törlöm az üzeneteket.



19. ábra Összes játékos adatainak módosítói

3.6 Hibák mentése

Hibák mentését igyekeztem úgy alakítani, hogy mindig egyértelmű legyen, hol történt a hiba és ne kelljen keresgélni kulcsszavak alapján. Ennek megfelelően létrehoztam egy `PuzzleExceptionLogger` osztályt, melynek feladata az alkalmazás saját kivételeinek mentése, valamint egy `UnhandledExceptionLogger`-t, aminek a feladata az olyan kivételeknek a mentése, amelyek nem az alkalmazás sajátjai. Végül egy újabb osztályt is létre kellett hoznom, ami nem kivételeket ment. Az `InValidModelLogger` osztályt azokban az esetekben használom, amikor a felhasználótól érkező adatok formája nem felel meg az elvártnak. Ilyen eset nem fordulhat elő a kliens alkalmazások használatával, de persze lehetséges ilyeneket külső eszközökkel létrehozni.
Minden esetben fájlba mentem a hibák adatait, amelynek helye konfigurálható.

3.7 Kliens alkalmazások tervezése

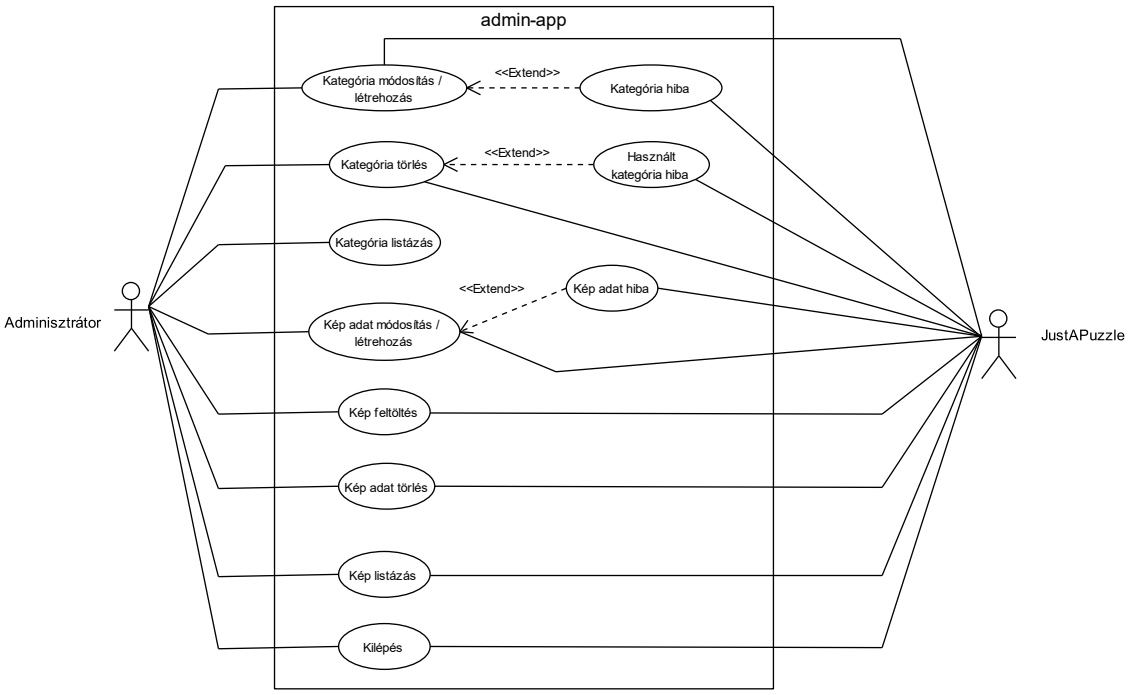
A kliens alkalmazásokat funkciók alapján bontottam modulokra. Mindkét alkalmazást Angular6 felhasználásával készítettem, és az ehhez kapcsolódó ajánlásokat követve alakult ki a struktúra. A nagyobb funkciók részére külön modulokat hoztam létre, azokon belül komponensek jelenítik meg az adatokat, figyelik a felhasználó akcióit és reagálnak ezekre. A komponensek egymással és a szerverrel service osztályokon keresztül kommunikálnak. Az alkalmazás konfigurációját az `AppModule` osztály végzi el, itt kell regisztrálni a többi modul osztályt, a kezdő komponenst és az olyan komponenseket, amelyeket teljes alkalmazásban használok (pl. navigáció).
Egy funkció modul mindig tartalmaz egy modul osztályt, ahol a modul komponenseit regisztrálom, valamint egy routing modult, ahol a modul útvonalait konfigurálom. A kiszolgáló osztályok a modul könyvtárába kerülnek közvetlenül. Minden komponens kap

egy saját könyvtárt, amiben a stílus, html, komponens osztály és a komponenshez kapcsolódó teszt fájlok találhatók.

3.8 Adminisztrátor alkalmazás

Egy adminisztrátort belépés után rögtön az adminisztrátor alkalmazás oldalra irányítom, ahonnan egy linket követve tud átlépni a játékos oldalra.

A kategória modulban külön komponense van a kategóriák listázásának, a kategória létrehozásnak és módosításnak. Az utóbbi két komponens ugyanazt a nézetet használja, mivel az adatellenőrzések ugyanazok mindkét esetben.



20. ábra Adminisztrátor alkalmazás use case diagramja

A `CategoriesService` osztály valósítja meg a szerverrel való kommunikációt. Külön könyvtárban találhatók az osztályok, amelyek megadják az adatok formáját, illetve az ezekből származtatott hibaosztályok. A hibákkal kapcsolatos visszajelzéseket az alkalmazás szempontjából külső osztályok végzik. A hibaosztályok példányait alapértelmezett adatként használom a komponensekben, így az alkalmazás nem áll meg csak azért, mert hiba történt.

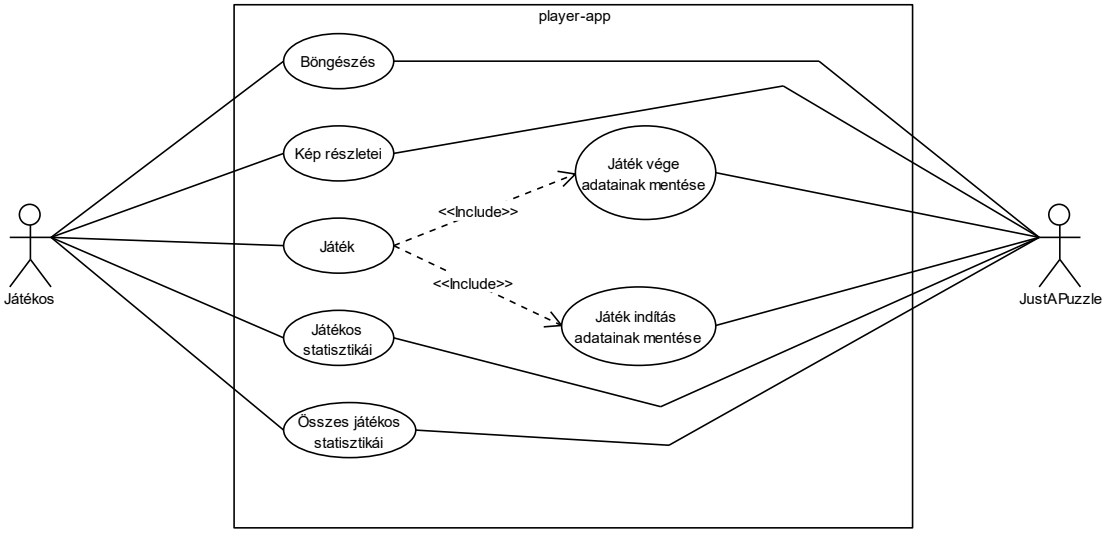
A képek modulban a létrehozás és frissítés komponensek esetében már mindkét komponensnek van nézet fájlja. Az elvégzendő ellenőrzések itt is megegyeznek a két esetben, de itt már több adatról van szó, ezért egy külön beviteli komponenst használnak fel. Ebbe a beviteli komponensbe ágyaztam be a fájl feltöltés komponenst, ami csak akkor

jelenik meg, ha már a módosítás komponenst látja a felhasználó. Minden kép komponenst a `PicturesService` szolgálja ki.

A fő alkalmazás modulban található a navigációs komponens, a listázás linkek mellett itt található a képek létrehozása link is. Ide helyeztem el, mivel ez a leggyakrabban használt funkció, emellett itt van a kilépés gomb is, ami egy komponens a közös modulból.

3.9 Játék alkalmazás

A játék alkalmazás a teljes alkalmazás legfontosabb része. Itt használom fel a korábbi részekben létrehozott adatokat és persze ezzel tudnak játszani a játékosok. Ezt a részt három főmodulra bontottam fel: böngészés, játék és a statisztikák megjelenítése.



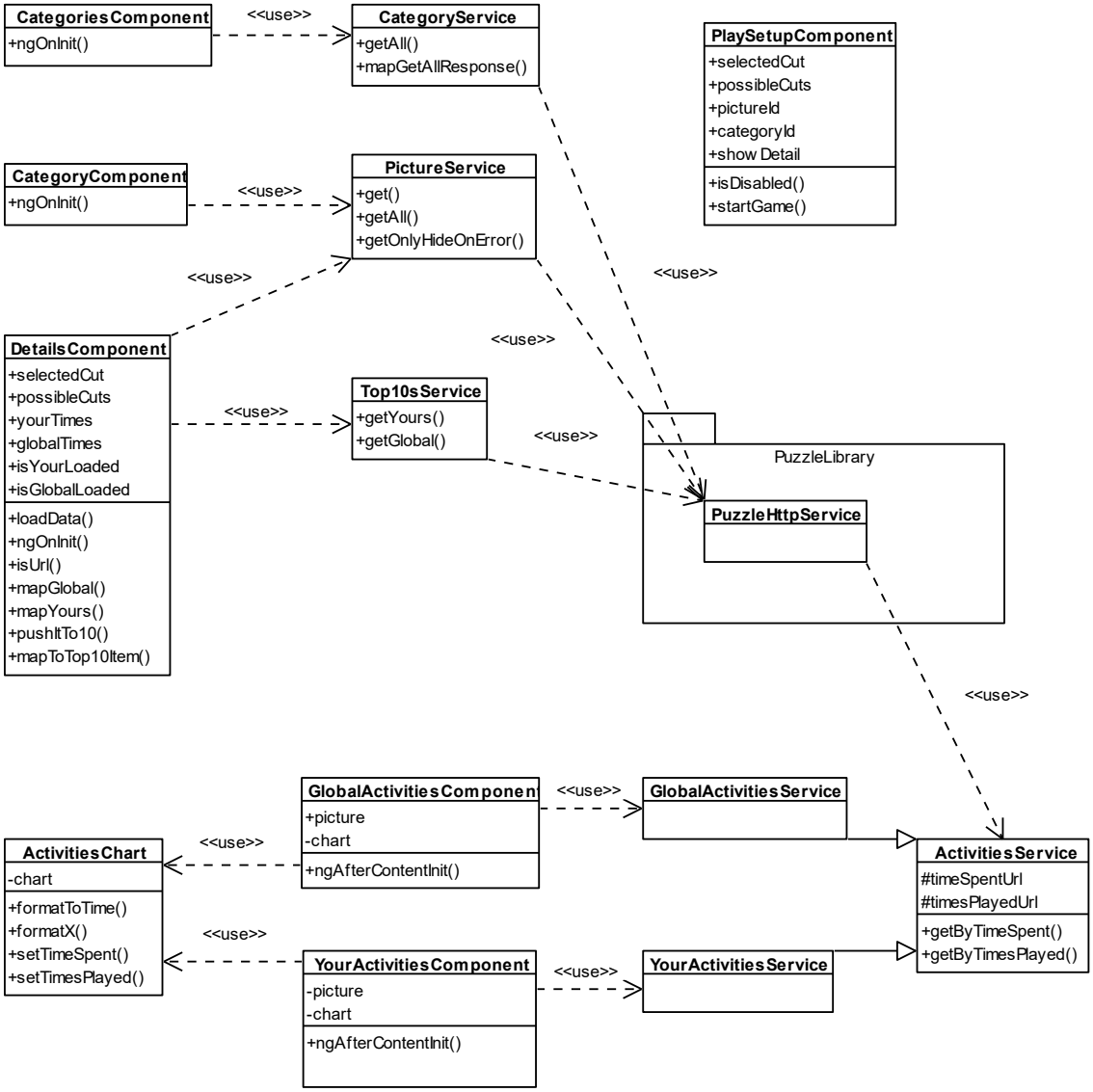
21. ábra Játékos alkalmazás felhasználói esetei

3.9.1 Böngészés

Böngészésnél megjelenítem a kategóriák listáját, amihez a kinézetet a `CategoriesComponent` implementálja, ami a `CategoryService`-t használva tölti be a kategóriák listáját. A lista elemek tartalmazzák a kategória nevét, azonosítóját és egy url-t a kategória előnézeti képéhez. Addig amíg le nem töltődnek ezek az adatok egy töltést jelző kép jelenik meg a felhasználónak.

Egy kategória kiválasztása után a `CategoryComponent` veszi át az irányítást, ami megjeleníti a kategóriához tartozó képeket, amiket a `PictureService`-et használva tölt le. Amennyiben hiba történne visszairányítom a játékost a kategória választó oldalra. Minden kép alatt megjelenik a `PlaySetup` komponens, ami egybefogja a kép részleteihez tartozó `Details` gombot, a játék indításhoz szükséges felvágás választó

legördülő menüt és a `Play` gombot. A `Play` gomb megnyomása után a játékost átnavigálom a játék url-re, ami paraméterként tartalmazza a játék beállításait.
Például: play/A/cut/BxC, ahol A egy adatbázis azonosító a képhez, B az oszlopok száma, C pedig a sorok száma.



22. ábra Böngészés modul osztályai

A kép részleteit a `DetailsComponent` tartalmazza, amikor először navigál ide a felhasználó egy teljes oldalt töltés jelző kép takarja el addig amíg be nem töltődnek a kép adatai, amit a `PictureService` szolgáltat. Miután ezek betöltődtek megjelenik a játék indító komponens, ami ugyanaz, mint a böngészésnél, csak itt nem jelenik meg a `Details` gomb. A komponensnek része még a legjobb idők megjelenítése is, ezekhez az adatok `Top10sService`-ből érhetők el, miután a játékos kiválasztott egy felvágást. Az utolsó 30 nap adatait megjelenítő komponenseket ide illesztettembe.

Külön komponenst kapott a játékos és az összes játékos adatainak megjelenítése saját kiszolgáló osztályokkal. A grafikon megjelenítését mindkét esetben az `ActivitiesChart` osztály végzi a c3 javascript csomag felhasználásával. Az osztály paraméterként kap egy azonosítót, amely megmondja melyik html elemhez kell kötni a grafikont. Az x tengelyen mindig a napok jelennek meg a bal y tengelyen az eltöltött idő, jobb y tengelyen a játszott játékok száma jelenik meg.

3.9.2 Játék elemek alapformájának terve

Minden egyes játékhoz fel kell darabolnunk a kiválasztott képet kisebb darabokra a megadott felvágás szerint és minden kisebb darabhoz létre kell hozni a megfelelő kitüremkedéseket vagy bevágásokat. Egy-egy ilyen kitüremkedés vagy bevágás a kisebb darabok valamelyik oldalának közepén helyezkedik el és körülbelül a harmadát veszi el. Ha a kép magasságát felosztom a függőleges felvágás szerint, majd ezt osztom hárommal kapunk egy blokk magasságot, amelyből három darab megadja egy elem magasságát, amelynek egyértelműen van egy közepe a bevágáshoz vagy kitüremkedéshez. Hasonlóan járok el a vízszintes felvágással is, így kapom a blokk magasságot. Ezek alapján a kis darabképeket már létre tudom hozni.
A következő lépés a kitüremkedések és a bevágások meghatározása. Rövid elemzés után látható, hogy ha a felvágásra úgy gondolok, mint egy mátrixra, akkor egy elemnél ezek meghatározhatók a következő logikát követve:

- Ha az oszlopindex * sorindex osztható 2-vel, akkor az elemnek jobb és baloldalon kitüremkedése van, felül és alul pedig bevágása
- Ha az oszlopindex * sorindex nem osztható 2-vel, akkor az elemnek jobb és baloldalon bevágása van, felül és alul pedig kitüremkedése van
- Ha egy elem sorindexe 0, akkor annál az elemnél nincs felső bevágás vagy kitüremkedés
- Ha egy elem sorindex-e megegyezik a függőleges felvágás-1-gyel, akkor az alsó oldala szintén üres
- Ha az oszlopindexe 0 vagy a vízszintes felvágás-1, akkor a bal, illetve jobb oldala üres

Ha a fenti szabályokat sorban végigfuttatom egy elemre, akkor megkapom a megjelenítendő alapformáját és ebből a szélességüket és a magasságukat is. A szélesség

és magasság meghatározásához, azonban még szükség van a kitüremkedések illetve bevágások méretének meghatározásához. Abban az esetben, ha úgy választom, hogy a méret mindig blokk magasság vagy szélességgel megegyező méretű, attól függően függőlegesen vagy vízszintesen torz összekapcsoló részeket eredményezne. Emiatt úgy döntöttem, hogy mind a szélességhez, mind a magassághoz a blokk magasságot használom.
A szélességre és a magasságra azért van szükség, mert az alapformával osztom fel a képet kis darabokra, de ha kitüremkedés van akkor az előző és a következő kis képből is szükség van egy kis darabra. A szélességet és magasságot még egy tulajdonság befolyásolja ez pedig a játéktér mérete, amit majd a böngésző ablak mérete határoz meg, ez alapján fogom átméretezni a képet, hogy az a játéktérnek csak a negyed részét foglalja el.
Ezt a funkcionalitást a `PieceVisual` osztály fogja implementálni, továbbá ennek az osztálynak lesz majd referenciája a `PieceLogic` osztály elemhez tartozó példányára is.

3.9.3 Játék elemek alapformájának implementációja

A játék elemeit a `PieceFactory` osztály hozza létre és konfigurálja. Ehhez kap a konstruktorban egy már átméretezett képet, amiből ki tudja számolni a blokk szélességet és magasságot. A számított adatok egy konfigurációs osztályba kerülnek, mert ezekre később az elem logikának is szüksége van. Végigiterál a sorokon és oszlopokon és minden cellához hozzárendel egy vizuális elemet, majd ezekhez létrehozza a kisebb képeket, amelyeket data url-ként ad át. A nagy kép feldarabolását a canvas html elemmel végzi el, majd létrehozás után beméretezi az elem szélesség és magasság szerint. Ezután a sor és oszlop index alapján meghatározza, hol kell majd lennie a kiskép bal felső sarkának az eredeti képen. Ezekkel az adatokkal és a képpel meghívja a canvas-hoz tartozó context drawImage metódusát. Végül a canvas to DataUrl metódusának meghívásával kapjuk a kívánt kisképet.
A `PieceVisual` osztály példányosításkor kiszámolja a kapcsolat elemek állapotát (in, out, empty), ezekből lehet majd később számolni az elem szélességet és magasságot, valamint az elemek maszkolásánál is ezekre az adatokra lesz szükségünk.

3.9.4 Játék elemek tervezett maszkolása

Ezen a ponton már meg vannak a kis téglalapok, amelyek az elemekhez adják a háttérképet, bár ez még nem hasonlít egy igazi puzzle elemre. A játéktér felületéhez és a

játékhoz először a canvas elemet terveztem használni, azonban hamar kiderült, hogy nincs olyan megoldás, ahol az egér mozgását késleltetés nélkül tudja követni a megjelenítés. Találtam azonban olyan megoldást, ahol svg elem esetén ez késleltetés nélkül történik meg. Így a korábban létrehozott kis képeket egy svg elemhez adom hozzá image elemként és egy clippath-t adok hozzá, ami megfelelően maszkolja.
Egy ilyen clippath legegyszerűbb puzzle elemnél is egy hosszú string-et eredményez, ezért a path létrehozását 4 részre bontom. Felső, jobb, alsó és végül alsó részekre, majd ezekből alakulnak ki az osztályok, amelyeknek csak a hozzájuk tartozó részt megrajzolja. Minden ilyen osztálynak szüksége van az elemhez tartozó `PieceVisual` példányra, de nem minden részére. Ezért létrehozok interfészeket, amelyek csak azt a részt jelenítik meg a `PieceVisual`-ból amire az adott útvonal készítőnek szüksége van. Az útvonal létrehozásához szükség van blokk szélességre és magasságra is, ezt a játék konfiguráció beinjektálásával oldom meg.

3.9.5 Játék elemek maszkolásának implementációja

Először a `PieceVisual`-hoz kapcsolódó interfészeket hoztam létre, ezek mindegyike 3 függvényt tartalmaz, ami megmondja az adott oldal üres-e, kitüremkedést illetve bevágást tartalmaz-e, ezeknek az implementációja minden esetben nagyon egyszerű, csak eltakarják, miként ábrázolom ezeket a `PieceVisual`-ban. Utána létrehoztam az útvonal készítő osztályokat, amelyekben minden rész lépéshez létrehoztam egy-egy metódust, amelyek a következő elgondolást követik:

- ha az oldal üres, akkor csak egy egyenes vonalat húz
- egyéb esetekben húz egy vonalat az oldal közepéig – blokk magasság/4
- majd attól függően, hogy kitüremkedés vagy bevágás egy vonalat befelé vagy kifelé
- egy félkört, aminek sugara a blokk magasság / 2
- egy vonalat, aminek hossza blokk magasság / 4
- egy az előzővel ellentétes félkört
- egy vonalat, amivel visszatérünk a kezdő vonalra
- és végül befejezi az oldal vonalát

Tervezéskor úgy, gondoltam a részlépéseket private elérhetőséggel valósítom meg, de végül ezek public-ként lettek megvalósítva. Ezzel a változtatással a tesztek írását

segítettem elő, mert így minden részlépésre tudtam tesztet írni és a tesztek utófeltételei is egyszerűbbek lettek.
Az osztályok egy oldalt a draw metódus meghívása után rajzolnak meg, és a draw metódust a `PieceVisual` hasonló nevű metódusa hívja meg. A maszk rajzolásánál relatív koordinátákat használtam, ezért a sorrend (felső, jobb, alsó, bal) nem változtatható meg.

3.9.6 Játék elemek logikájának tervezése

A játék elemek logikájának tervezésekor először arra gondoltam, minden egyes elemnél tárolom a kapcsolóelemek státuszát. Ez azonban több problémát okozott volna, mint amennyit megold. Minden esetben meg kellett volna nézni, mely elemek csatlakoztak már és azt is szerettem volna elérni, hogy az összekapcsolt elemek egyszerre mozogjanak, ami megint rengeteg új vizsgálatot hozott volna. Végül annak ellenőrzéséhez is meg kellett volna vizsgálni az összes elemet, hogy minden egyes kapcsolóelem kapcsolt állapotba került-e.
Az jobb megoldás szerint minden elem beletartozik egy halmazba létrehozáskor és amennyiben össze tudom kapcsolni őket drag and drop után, akkor közös halmazba kerülnek. Ezzel egyszerűsödött az elemek csoportként mozgatása, mivel nem kell megnézni minden elemre, hozzá van-e kapcsolva a mozgatott elemhez, hanem csak megnézem ugyanabban halmazban van-e és ha igen, akkor mozgatom. A játék végének ellenőrzése is egyszerűbb lett, ha már csak egyetlen halmaz van akkor a felhasználó kirakta a képet.
Az elem logikája tartalmazza az elem koordinátáit a játéktéren, erre szükség van a közelség ellenőrzéshez, ami az alapja az összekapcsolhatóság ellenőrzésének. Az összekapcsolás határait a `ConnectLimits` osztály határozza meg.
Az összekapcsolást mindig a húzott elemre nézem meg, elég közel-e van a másik elemhez, míg a másik elem meghatározása majd a játék logika feladata lesz. Ha összekapcsolható akkor közelebb húzom egymáshoz az elemeket (és persze a halmazaikban levőket is). A tényleges összekapcsolás, ami itt az egy halmazba kerülést jelenti szintén a játék logika feladata.

3.9.7 Játék elemek implementációja

Ezt a megoldást a `PieceLogic` osztály implementálja, ahol minden irányra egy külön metódus vizsgálja, hogy elég közel van-e a két elem. Ezek igaz értéket adnak vissza, ha teljesül a feltétel, utána a játék logika hívja meg a mozgató metódusokat, hogy pontosan illeszkedjenek. A `ConnectLimits` osztályban több határt is be kellett vezetnem, az alsó és felső határokat a függőleges és vízszintes vizsgálatokhoz, illetve kisebb határokat az ellentétes irányoknál való ellenőrzéshez. Erre azért van szükség, mert a pont, ami meghatározza egy elem pozícióját az mindig a bal felső sarok. Ha egy elemet közel mozgatunk egy másikhoz, ami tőle jobbra helyezkedik el összekapcsolhatóság szempontjából, akkor a vízszintesen a fogópontok különbsége vízszintes irányban egy háromszor blokkszélesség közeli érték kell legyen, viszont a függőleges irányban majdnem egy vonalban kell lennie.
Az implementáció során azzal a problémával is meg kellett küzdenem, hogy bár úgy terveztem minden elemet, mint ami háromszor blokk magasság magas és háromszor blokk szélesség széles, abban az esetben, ha kitüremkedés van a bal oldalon vagy az elem tetején akkor a fogópont, aminek koordinátája meghatározza az elem helyét, egy blokk magassággal több, mint amire számítottam. Ezt az értéket egy különbség pont tartalmazza minden logikában, amit az összekapcsolhatóság számításnál figyelembe veszek.

3.9.8 A játék logika

A játék logika feladata, hogy tárolja a logikai elemeket, valamit a logikai elem halmazokat, egy húzott elemhez, illetve húzott halmazhoz. Továbbá meghatározza a húzás végén, mely elemekre érdemes ellenőrizni, hogy össze tudjuk-e kapcsolni és végül frissítse a játék történetét. A játék logika először mátrixba helyezi el a logikai elemeket a képen elfoglalt oszlop, sor koordinátáik alapján, ezzel a lehetséges elemek gyorsan meghatározhatók az ellenőrzéshez. Egy adott elemhez a felső lehetőség mindig egy sor koordinátával feljebb van (sor - 1), ugyanabban az oszlopban, a bal oldali lehetőség egy oszlop koordinátával balra (oszlop - 1), az alsó lehetőség egy sorral lejjebb (sor + 1) és végül a jobb lehetőség a jobboldalán lévő oszlopban (oszlop + 1). Így ahhoz, hogy megkeressem a lehetőségeket csak a megfelelő koordinátákon kell keresni az elemeket.
Minden húzás után a `checkPieces` metódus hívódik meg, ami egy játék eleme logikát kap paraméterként. Az ellenőrzést nem csak az adott elemre végzem el, hanem a játék

elem halmazának minden elemére, így történik meg az összes lehetséges összekapcsolás. Az ellenőrzés egy elemre abból áll, hogy minden irányt végignéz a kód, lehet-e ott összekapcsolás, ezt a játék elem logika megfelelő metódusa végzi. Ha lehetséges akkor egymáshoz illeszti az elemeket, majd közös halmazba helyezi őket.
Mikor minden elemre lefutott az ellenőrzés, akkor azokat a halmazokat, amelyekre már nincs szükség kiveszem az összes halmaz listájából és gondoskodom arról, hogy minden elemnek, amely most kapcsolódott össze, annak ugyanaz legyen a halmaza.
Ezután húzom majd vissza a játéktérről kikerült elemeket, megkeresve a vízszintesen, illetve függőlegesen legjobban kilógó elemeket, majd minden a húzott halmazba tartozó elem koordinátáját módosítom a kapott különbséggel.
Legutoljára történik a játéktörténet adatainak frissítése egy új játék lépés felvételével, amihez csatolom mely elemeket húzta a játékos és mely elemek lettek összekapcsolva a lépés során.
A játék logikát a `GameLogic` osztály implementálja, a konstruktorban inicializálom az elemek mátrixát és a kezdeti halmazok listáját. A játéktörténet készítéséhez az ellenőrzéséhez csinálok egy új tömböt, ami az eredeti húzott halmaz elemeit tartalmazza. Ha történik összekapcsolás az elem halmaza megváltozik és az lesz az összekapcsolt elemek halmaza. A lépést magát a `GameHistoryService` illeszti hozzá a korábbi lépések listájához. A játék végének ellenőrzését az `areAllPiecesConnected` metódus hívásával lehet megtenni, ezt majd a játék építő osztály hívja meg.

3.9.9 Közvetítő osztályok

Az alkalmazás osztályai közvetítő osztályokkal kommunikálnak egymással, Angular6 framework használatával létre lehet hozni service osztályokat, amelyek singleton-ként működnek, minden olyan osztály, ami ilyen feladatot lát el service-ként hoztam létre. Ezek minden olyan osztály példányba, amit a framework hoz létre, automatikusan beinjektálódnak. Azok az osztályok, amelyeket nem a framework hoz létre, konstruktor paraméterként kapják meg ezeket.
Egy játékot, mindig a `GameStartService` indít, amelynek a feladata, hogy példányosítsa a játék építő osztályt, és mentse az adatokat az aktuális játékhoz, amit majd újra játszás esetén fogok majd felhasználni.
Arra is szükség van, hogy az eltelt időt több helyen tudjuk kezelni. Egyszer a kiíratáshoz, ami nem magában a játék komponensben történik, hanem a navigációs sávon, másrészt a

játék történet vezetéséhez is. Ezt a funkcionalitást a `TimeService` valósítja meg. Játék indításkor elmenti a kezdeti időpontot, majd ez alapján előállít egy formázott szöveget, amit meg lehet jeleníteni. Végül a vég időpontot is itt tárolom.

A segítségék megjelenítéséhez is service-t (`HintService`) használok, ennek szintén az az oka, hogy ennek kezelőfelülete is navigációs komponensen érhető el. Ez a service tartja számon hány segítségét kért a játékos, itt keresem meg azokat az elemeket, amelyek összekapcsolhatók, ilyen segítségkérésnél egyszerűen csak megkeresem az első elemet, amelynél még nincs minden kapcsolható elem hozzákapcsolva. A kapcsolható elemekből is kiveszem az elsőt, majd mindkettőt megjelölöm egy fekete keretű fehér körrel. A másik segítség, amit ez az osztály valósít meg, hogy már a felhasználóktól érkezett kérésként, a kis elem halmazokat nagyobbak el tudnak takarni, emiatt a játék vége felé közeledve, ezek nehezen megtalálhatók, ezért a játék elemeket sorba rendezem a legnagyobbtól a legkisebbig, majd sorban mindegyiket a legfelső elemmé teszem így a legkisebb halmazok legfelülre kerülnek.

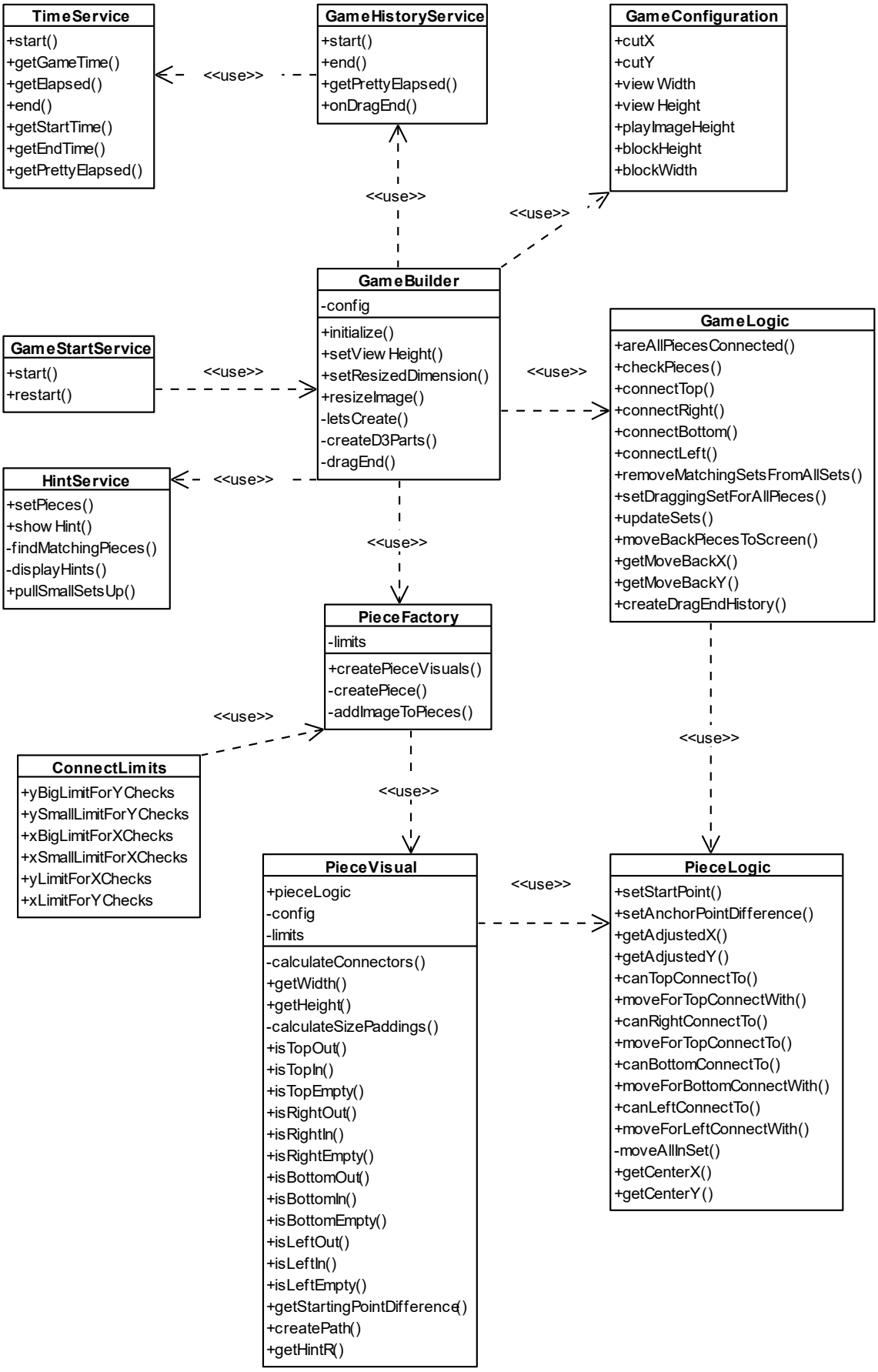
Szintén a felhasználás során derült ki, a játéktér háttér színét nem lehet úgy megválasztani, hogy az minden egyes képhez megfelelő legyen. Egy sötét háttér jó a világos képekhez, egy világos pedig a sötétekhez. Ezért hoztam létre egy háttérszín változtató service-t (`BackgroundBrightnessService`), aminek segítségével a felhasználó tudja állítani ezt.

A játék történet vezetését is service-szel oldottam meg (`GameHistoryService`). Ennek feladata a játék kezdetén üzenetet küldeni a szerver alkalmazásnak a játék megkezdéséről. Az üzenet tartalmazza, hogy a felhasználó melyik képpel és. milyen felvágással játszik, valamint és játék kezdetének időpontját. Válaszként kapja a játékhoz tartozó azonosítót, amit a játék végén használok fel. Játék közben ez az osztály tárolja a játék lépéseit és egy húzás kezdetén létrehoz egy új lépés (`Step`) példányt. Ez rögtön tárolja, mikor kezdődött a húzás, majd a húzás végén hozzáadja, mely elemeket mozgatta a játékos, az elemeket, amelyek ennek a húzásnak a hatására kapcsolódtak össze és végül azt is mikor fejeződött be a húzás. Játék végén ezeket az adatokat elküldi a játék befejezésének időpontjával együtt a szervernek, ami ez alapján frissíti a játékoshoz kapcsolódó és összesített statisztikákat.

A játék végén, illetve segítségként is, megjelenítem a kirakandó képet, ehhez szükség van arra, hogy több komponens is elérje a kép url-jét ezt a `GamePictureUrlService`-szel oldom meg.

3.9.10 Játék építő

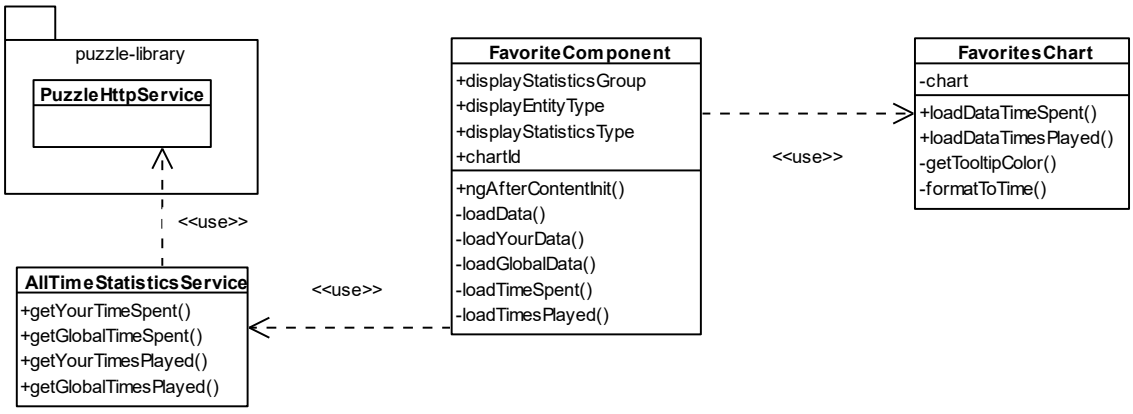
A játék építő (`GameBuilder`) feladata, hogy összeállítsa magát a játékot, ebben az osztályban kapcsolódik össze a megjelenítés és a logika.
Játék kezdetén ebben az osztályban határozom meg a kirakandó kép megjelenítési méretét, amely a megjelenített játéktér magasságnak a fele és a szélesség pedig ezzel arányos. Ezután méretezem át a képet egy canvas elem segítségével, így az utána következő lépések már ezzel az méretezett képpel dolgoznak. Ebből hozza létre a `PieceFactory` a megjelenítendő elemeket és a hozzájuk kapcsolt logikát. A kapcsolt logikákat kapja meg a `GameLogic` osztály, és a megjelenítendő elemeket használja a `createD3Parts` metódus.
Ebben a metódusban először a húzás kezelést konfigurálom, amihez létre kell hozni egy függvényt, mely kezeli a húzást és annak végét is. Egy húzás kezdetét és annak folyamatát egy függvény kezeli, ahol először mindig felső elemmé teszem a húzott elemet és a halmazában lévő elemeket, majd módosítom az elemek koordinátáit a húzás értékével. Az első ilyen esemény jelenti azt, hogy a húzás megkezdődött, ezért ez hozza létre a következő játék lépést is. A húzás végén lefutó függvényben ellenőrzöm, az elemek hozzákapcsolhatók-e újabb elemekhez és miután ez megtörtént, azt is, hogy a húzás után minden elem össze van-e kapcsolva.
Az elemek megjelenítését egy svg elem végzi el, amit a d3 library segítségével módosítok. Először felhelyezem az svg elemet a játéktérre, azzal megegyező méretben és a korábban beállított háttérszínnel. Ezután ebbe kerülnek bele az elemek maszkjai, amiket a sor és oszlop indexek azonosítanak. Ezek az svg elemek szintjén, mint clipPath jelennek meg és tartalmaznak egy path elemet, amit a `PieceVisual` állít elő.
Végül a részképeket hozzáadom az svg-hez, és a maszkokat a sor és oszlop index alapján kapcsolom a részképekhez. A kezdeti pozíciót az elem logikai objektumokból kapják, ehhez az image elem transform attribútumnak kell egy megfelelő translate függvény értéket adni. A képet dataurl-ként tartalmazza a PieceVisual objektum, ezt kell értékként adni az xlink:href attribútumnak. A korábban létrehozott húzáskezelő függvény hozzákapcsolása teszi őket mozgathatóvá, amit call függvény használatával teszek meg.
Ezután még szükség van arra is, hogy minden image elemhez és az svg-hez is hozzárendeljek egy mousedown eseménykezelőt, ami a segítségként felhelyezett köröket tünteti el.



23. ábra A játék csomag osztályai

3.9.11 Statisztikák

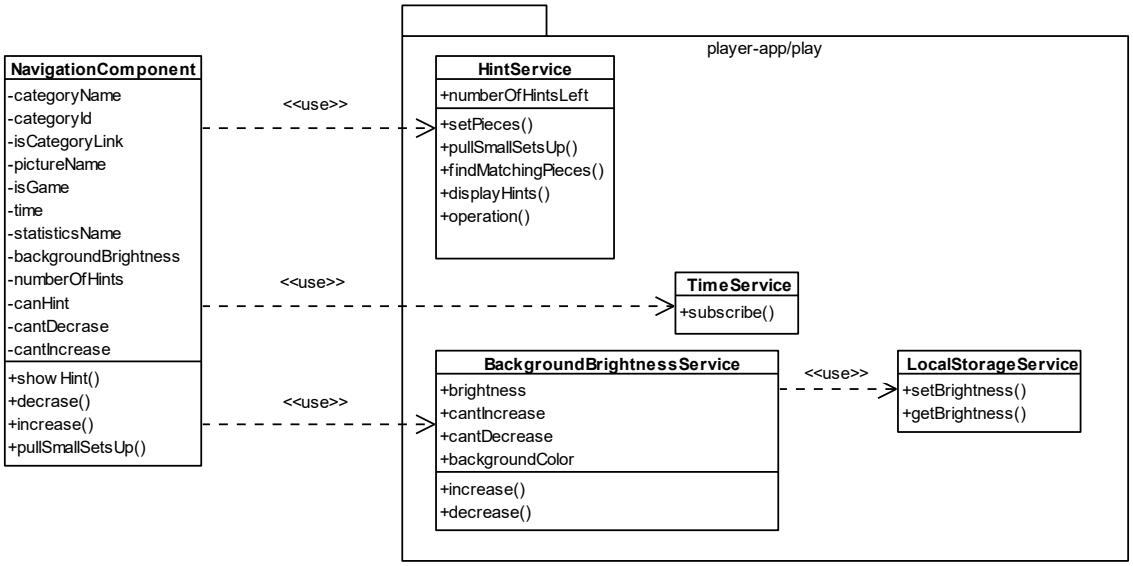
A statisztikákban kétfajtát különböztetek meg: a játékos saját adatai és az összesítettek, mivel az adatok formája egyező, ezért a létrehozott komponensek és a service osztályok paraméterként kapják meg, hogy éppen melyikre kíváncsi a felhasználó.
A statisztikák megjelenítését egy komponens fogja össze (`StatisticsComponent`). Ebbe elsőként az összefoglaló komponenst illesztem, ahol szövegesen jelenítem meg az elkezdett, illetve a befejezett játékok számát, a megtett lépések számát, a játékkal eltöltött időt és ezeknek átlagát, ez a komponens a `SummaryService`-t használja az adatok eléréséhez.
Utána következnek a kedvencek diagramjai, ahol a megjelenítéshez mindegyik egy komponenst használ (`FavoriteComponent`). A komponens az `AllTimeStatisticsService`-t használja az adatok feltöltéséhez, majd a `FavoriteChart` osztály generálja a diagramokat. Itt két különböző diagram fajtát hoztam létre: egyet az eltöltött időhöz, egy másikat pedig a játékok számához. A diagram generálás maga mindkét esetben hasonlón történik. Először meg kell adni mely elem jelöli ki a diagram helyét (`bindTo`), majd létre kell hozni egy objektumot (`data`), ami tartalmazza az adatokat (`json`). Ezután meg kell adni, hogy az adatok hogyan jelennek meg az x tengelyen (`keys`), a diagram típusát (`type`) és meg kell adni egy a színeket meghatározó függvényt (color). Az egész konfigurációt ezután át lehet adni a c3 library `generate` függvényének, ami ez alapján megjeleníti a diagramot. Négy különböző diagram jelenik meg a kedvenc kategóriák eltöltött idő és játékok száma szerint, illetve ugyan ezek kedvenc képek szerint.



24. ábra A statisztikák modul osztályai

3.9.12 Navigációs komponens

A játék alkalmazás navigációs komponensén jelennek meg a felhasználó által elérhető funkciók. A komponensen különbözőképpen jelenik meg aszerint, hogy éppen fut egy játék vagy sem. Ha nem fut játék, akkor a böngészés, statisztikák és a kilépés gomb foglalja el. Mikor fut egy játék, akkor itt jelennek meg a következő gombok: az eltelt idő, a háttér színének állítását végző gombok, a segítségek menü, melynek elemei a játszott kép megjelenítése, az összekapcsolható elemek kijelölése és a kisebb esetleg elfedett elemek felemelésének lehetősége. Emellett található a főmenü, aminek elemei azok, amelyek a nem fut játék esetben megjelennének.



25. ábra Navigációs komponens függőségei

3.10 Közös kliens modul

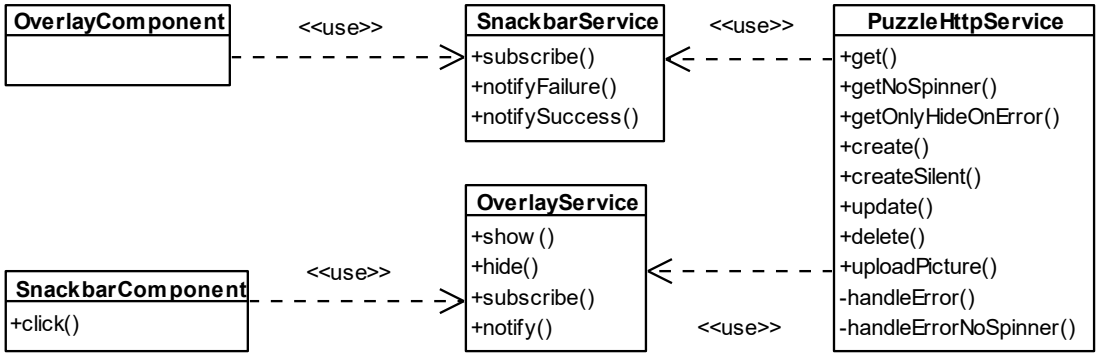
A közös kliens modulba kerültek azok a funkcionalitások, amiket mindkét kliens alkalmazás használ.

Ezek közül a legfontosabb a `PuzzleHttpService`, a használt Angular6 library-ben adott egy megoldás, amivel http kéréseket hajthatunk végre. Viszont minden kérésnél egyenként kell beállítani, hogy miként kezelje a hibákat, melyek mindkét alkalmazásban hasonlóak, ezért lehet általánosan kezelni ezeket. A http kérések kezeléséhez tartozik még a visszajelzés is a felhasználó felé, hogy most épp történik-e valami. Az alkalmazásokban visszajelzés alapján az alábbi típusokat lehet meg különböztetni: blokkolni kell a felhasználó felületet addig amíg nem fejeződik be a kérés; ezzel párhuzamosan olyanok, amelyeknél nem szabad blokkolni; olyan kérések, amelyeknek a sikerességét külön kell

jelezni; illetve amelyekét nem; valamint minden esetben jelezni kell hibákat. A service ezekhez típusokhoz kapcsolódó kéréseket valósítja meg.

A felhasználói felület blokkolásához az `OverlayService`-t használom, ehhez kapcsolódik egy komponens, ami megjelenít egy várakozás jelzőt, ennek az állapotát változtatja a service. A sikeres és sikertelen kéréseket a `SnackbarService` végzi el, ehhez is kapcsolódik egy komponens és ebben az esetben is ennek állapotát változtatja a service.

Ide került még a kilépés komponens is, mivel maga kilépés folyamata teljesen ugyanaz mindkét fajta felhasználó számára.



26. ábra Puzzle library osztályai

3.11 Az alkalmazás fordítása

Mivel több technológiát is használok az alkalmazásban, ezért a teljes fordítás is több lépésből áll össze. A kliens alkalmazások fordításához telepíteni kell nodejs-t (v8.11.4), npm-et (5.6), és az angular-cli-t (6.1.5).

Miután ezek telepítése megtörtént, először a közös kliens modult kell lefordítani. Ehhez be kell lépni a puzzle-library könyvtárba, helyre kell állítani az npm csomagokat az `npm install` paranccsal, majd az `ng build puzzle-library –prod` parancs fordítja le a modult. A kliens alkalmazások fordítása megegyezik. Be kell lépni a fordítandó alkalmazás könyvtárába, helyre kell állítani a csomagokat, és az `npm run buildprod` fordítja le a klienst.

A szerver fordításához a .Net Core Sdk-t (2.1) kell telepíteni, a `dotnet build` parancs helyreállítja a szükséges csomagokat és lefordítja az alkalmazást.

3.12 Tesztelés

A szerver, illetve a játékos alkalmazáshoz készítettem automata egységteszteket. Az alkalmazás többi részét kézi end-to-end tesztekkel teszteltem.

3.12.1 A játékos alkalmazás tesztjei

A TypeScript tesztekhez jasmine teszt csomagot és karma teszt futtatót használtam, a tesztek futtatáshoz ezen kívül szükséges egy telepített FireFox böngésző. A teszt fájlok `.spec.ts` kiterjesztésűek és a tesztelt fájlok mellett találhatóak. A futtatásokhoz először le kell fordítani a közös kliens modult.

3.12.1.1 A maszkolások tesztelése

A maszkolásokat a `TopPathMaker`, `RightPathMaker`, `BottomPathMaker` és `LeftPathMaker` osztályok végzik. A hozzájuk írt tesztjeim mind hasonlóak. Először tesztelem, hogy minden egyes részlépés az elvárt kimenetet hozza-e létre, majd azt, hogy a részlépések egymásutánja is az elvárt eredményt adja-e.

3.12.1.2 A játék elemek formájának tesztelése

A játék elem formáját a `PieceVisual` osztály határozza meg. Tesztet írtam arra, hogy a kitüremkedések és bevágások a sor és oszlopindex alapján megfelelően számolódnak, illetve az, hogy az elem a szélek valamelyikén helyezkedik el megfelelően módosítja ezt. Továbbá azt is tesztelem, hogy az elem szélessége is helyesen számítódik.

3.12.1.3 A játék logika tesztelése

A játék logikánál tesztelem, hogy a játéktérről kilógó elemek vissza kerülnek a játéktérre, minden húzás után a lépés adatai bekerülnek a játéktörténetbe, a játék véget ér, ha már csak egyetlen halmaz marad, és egy húzott elem minden olyan elemez hozzákapcsolódik amihez tud.

3.12.1.4 Az elem logika tesztelése

Az elem logikánál tesztelem, hogy alsó, felső, bal és jobb lehetőségek esetén az elemek összekapcsolhatók, ha elég közel vannak és hogy ugyan ezekben az esetekben az összekapcsolt elemeket egymáshoz illeszti. Az elem kezdő pozícióját is ez az osztály határozza meg erre is írtam tesztet.

3.12.2 A szerver alkalmazás tesztjei

A szerver alkalmazás tesztjeihez az xunit és a Moq csomagokat használtam fel. A szerver alkalmazás tesztjei a `dotnet test` paranccsal lehet lefuttatni, amit a server mappában kell kiadni.

3.12.2.1 Emailek tesztelése

Az email esetében tesztelem, hogy a megfelelő kivételeket dobja az alkalmazás a hibák esetén. Az emailek küldésénél az üzenet tartalmát és a küldő kliens működését ellenőrzöm, különös tekintettel arra, hogy a megfelelő linkek kerüljenek be az email megerősítésnél, illetve a jelszóváltoztatásnál.

3.12.2.2 A felhasználói fiók tesztelése

A felhasználói fióknál a framework osztályai végzik a beléptetést, jelszóváltoztatást és a regisztrációt. Itt ezért azt tesztelem, hogy az előellenőrzések során a megfelelő kivételekkel jelezi a hibát.

3.12.2.3 Játék tesztelése

A játék tesztelésénél külön teszteket írtam arra, hogy a játék kezdésnél és befejezésnél ellenőrzöm az adatokt. Majd a játék osztályra írtam teszt, hogy megfelelően használja-e a függőségeit, játék kezdésnél és befejezésnél az adatokat továbbadja mentésre, és meghívja a statisztika frissítő osztályok megfelelő metódusait.

3.12.2.4 Hiba mentés tesztelése

Itt minden hibamentési lehetőségre írtam teszteket, amelyek azt ellenőrzik, hogy a hiba adatai az elvárt formátumban és adatokkal kerülnek mentésre.

3.12.2.5 Kategória tesztelése

A kategóriánál tesztelem, hogy olvasásnál kivételt dob, ha nem létező kategóriát olvasna. Létrehozás és frissítés esetén, hogy a név egyedi legyen. Törlésnél pedig azt, hogy dob-e kivételt, ha a kategóriában van kép.

3.12.2.6 Képadat tesztelése

Képadat esetén tesztelem, hogy amikor minden képet próbál lekérni, akkor történik-e kivétel, ha nem létező kategóriát próbál listázni és azokat az eseteket is amikor vissza kell

térni értékkel. Írás esetén teszteket írtam az ellenőrzésekre és arra, hogy a helyes adattal hívja meg az íróosztály metódusait.

3.12.2.7 Attribútumok tesztelése

A kivételkezelő attribútumoknál tesztelem, a hibák mentését, az attribútum lekezeli a kivételt és helyes hiba választ küld vissza a kliensnek. A hibás adatokat kezelő attribútumnál csak a hiba mentését tesztelem.

3.12.3 End-to-End tesztek

3.12.3.1 Adminisztrátor belépése

Kezdő állapot:	Vannak olyan kategóriák, amelyeknél már van előnézeti kép és vannak olyanok is amelyeknél még nincs
Lépések:	A felhasználó belép egy adminisztrátor fiókkal
Várt eredmény:	Az adminisztrátor alkalmazás jelenik meg Minden kategória látszik a listában

3.12.3.2 Új kategória létrehozása

Kezdő állapot:	A felhasználó az adminisztrátor alkalmazásban van
Lépések:	Kattint a Categories linkre A megjelenő oldalon kattint az Add new category gombra Beír egy olyan kategória nevet, ami hosszabb, mint 100 karakter
Várt eredmény:	Megjelenik a hibaüzenet, hogy a kategória neve nem lehet hosszabb 100 karakternél
Lépések:	Beír egy már létező kategória nevet Kattint a Create gombra
Várt eredmény:	Jobb oldalt felül megjelenik a már létezik ilyen kategória hibaüzenet
Lépések:	Beír egy olyan kategória nevet, ami nem létezik még és kattint a Create gombra
Várt eredmény:	A Categories oldal jelenik meg, ahol már szerepel az új kategória

3.12.3.3 Új kép létrehozása

Kezdő állapot:	A felhasználó az adminisztrátor alkalmazásban van
Lépések:	Kattint a Create new picture linkre
Várt eredmény:	Megjelenik az új kép oldal
Lépések:	Beír egy olyan kép nevet, ami hosszabb, mint 100 karakter
Várt eredmény:	Megjelenik a túl hosszú név hibaüzenet
Lépések:	Beír egy rövidebb nevet, majd kattint a Create gombra
Várt eredmény:	Megjelenik a kép módosítás oldal és a lehetőség a kép feltöltésre
Lépések:	Kiválaszt egy képet majd kattint az Upload gombra
Várt eredmény:	Megjelenik a feltöltött kép előnézeti képe és a generált felvágások listája

3.12.3.4 Kép törlése

Kezdő állapot:	A felhasználó az adminisztrátor alkalmazásban van
Lépések:	Kattint a Pictures linkre, majd kiválaszt egy kategóriát A megjelent listán a Delete gombra kattint egy kép mellett
Várt eredmény:	A kép eltűnik a listából, miután megjelent a sikeres törlést jelző üzenet

3.12.3.5 Játékos belépése

Kezdő állapot:	Vannak olyan kategóriák, amelyeknél már van előnézeti kép és vannak olyanok is amelyeknél még nincs
Lépések:	A felhasználó belép egy játékos fiókkal
Várt eredmény:	A játékos alkalmazás jelenik meg Csak azok a kategóriák látszanak, amelynél már van előnézeti kép

3.12.3.6 Játék indítása

Kezdő állapot:	A felhasználó belépett a játékos oldalra
Lépések:	Kattint egy kategóriára
Várt eredmény:	Megjelennek a kategória képei
Lépések:	Klikkel egy kategóriára
Várt eredmény:	Megjelennek a kategória képei, alattuk felvágások legördülő menü tartalmazza a felvágásokat, a Play gomb nem aktív
Lépések:	Kiválaszt egy felvágást valamelyik képnél

Várt eredmény:	A Play gomb aktív lesz
Lépések:	Kattint a Play gombra
Várt eredmény:	Rövid idő eltelte után elindul a játék, a játéktéren a választott kép, a választott felvágás szerint jelenik meg

3.12.3.7 Játék

Kezdő állapot:	A felhasználó elindított egy játékot
Lépések:	Kattint egy elemre és a gombot nyomva tartva húzza
Várt eredmény:	A kép mozog az egér mozgásának megfelelően
Lépések:	Közel mozgatja az elemet egy csatlakozó elemhez, majd felengedi az engedi az egér gombját
Várt eredmény:	A távolság, ha volt az elemek között eltűnik és az elemek összekapcsoltnak néznek ki
Lépések:	Mozgatja a frissen összekapcsolt elemek egyikét
Várt eredmény:	A másik elem is mozog vele
Lépések:	Összekapcsolja az összes elemet
Várt eredmény:	Megjelenik egy modális ablak, melynek a tetején a játék ideje látható, alatta pedig a kirakott kép

3.12.3.8 Segítségek

Kezdő állapot:	A felhasználó elindított egy játékot
Lépések:	Kattint navigációs sávon lévő plusz gombra
Várt eredmény:	A háttér világosabb lesz
Lépések:	Kattint navigációs sávon lévő mínusz gombra
Várt eredmény:	A háttér sötétebb lesz
Lépések:	Kattint a Hints menü Show picture elemére
Várt eredmény:	Megjelenik egy modális ablakban a kirakandó kép
Lépések:	Bezárja az ablakot, majd kattint a Hints menü Hints left elemére
Várt eredmény:	Kijelölődik két elem, ami összekapcsolható, a Hints left értéke csökken
Lépések:	Egy elemet eltakar egy csoporttal, majd kattint a Hints menü Pull small groups up elemére
Várt eredmény:	Az eltakart elem az őt eltakaró elem fölé kerül

4 Irodalomjegyzék

[1] ASP.NET Core MVC Identity using PostgreSQL database, https://medium.com/@RobertKhou/asp-net-core-mvc-identity-using-postgresql-database-bc52255f67c4, 2017.07.23.

[2] Automated nginx proxy for Docker containers using docker-gen, https://github.com/jwilder/nginx-proxy, 2018.09.14.

[3] Circle Dragging I, https://bl.ocks.org/mbostock/22994cc97fefaeede0d861e6815a847e, 2018.03.05.

[4] Chart examples, https://c3js.org/examples.html, 2018.10.24.

[5] Cutting an Image into pieces through Javascript, https://stackoverflow.com/questions/8912917/cutting-an-image-into-pieces-through-javascript, 2018.03.05.

[6] Configuration Basics, https://github.com/serilog/serilog/wiki/Configuration-Basics,2017.02.18.

[7] Configuration in ASP.NETCore, https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration, 2017.09.05.

[8] File uploads in ASP.NETCore, https://docs.microsoft.com/en-us/aspnet/core/mvc/models/file-uploads, 2017.02.26.

[9] How TO - Overlay, https://www.w3schools.com/howto/howto_css_overlay.asp, 2018.03.06.

[10] How TO - Snackbar / Toast, https://www.w3schools.com/howto/howto_js_snackbar.asp, 2018.03.06.

[11] How to use the IOptions pattern for configuration in ASP.NET Core RC2, https://andrewlock.net/how-to-use-the-ioptions-pattern-for-configuration-in-asp-net-core-rc2/, 2017.02.23.

[12] HttpClient, https://angular.io/guide/http, 2018.11.10.

[13] Introduction to ASP.NET Core, https://docs.microsoft.com/en-us/aspnet/core, 2017.02.18.

[14] LetsEncrypt companion container for nginx-proxy, https://github.com/JrCs/docker-letsencrypt-nginx-proxy-companion, 2018.09.15.

[15] MailKit, Sending messages, https://github.com/jstedfast/MailKit#sending-messages, 2017.09.05.

[16] Modals, https://valor-software.com/ngx-bootstrap/#/modals, 2018.10.25.

[17] Sending Smtp Email on ASP.NET Core with MailKit, https://www.joeaudette.com/blog/2016/05/08/sending-smtp-email-on-aspnet-core-with-mailkit, 2017.07.30.

[18] Step by step: Serilog with ASP.NET Core, https://carlos.mendible.com/2016/09/19/step-step-serilog-asp-net-core/, 2017.08.06.

[19] Testing, https://angular.io/guide/testing, 2018.11.12.

5 Mellékletek

A program forrása, a futtatáshoz szükséges fájlok és a szakdolgozat pdf verziója megtalálható a szakdolgozat hátlapjára ragasztott papírtokban lévő DVD-n.

- A forráskód a “Forrás mappában található”, ez alatt a `clients` könyvtárban a kliensek forrásfájljai (Angular6 project-ek), a `server` mappában pedig a szerver fájljai (Visual Studio solution).
- A futtatáshoz szükséges fájlok a ”Futtatható” mappában találhatóak `justapuzzle.tar` illetve `docker-compose.test.yml`.
- A szakdolgozat pdf verziója a DVD gyökérkönyvtárában, ”Szakdolgozat_Gyetvai_Gabor.pdf” néven található meg.